

Contents

1 Alternate Odd and Even Nodes in a Singly Linked List	9
Source	16
2 Arithmetic Expression Evaluation	17
Source	19
3 Balanced expression with replacement	20
Source	26
4 Bank Of America (BA Continuum India Pvt. Ltd.) Campus Recruitment	27
Source	28
5 Bubble sort using two Stacks	29
Source	31
6 Check if stack elements are pairwise consecutive	32
Source	34
7 Check for balanced parentheses in an expression	35
Source	43
8 Check for balanced parenthesis without using stack	44
Source	46
9 Check if a given array can represent Preorder Traversal of Binary Search Tree	47
Source	52
10 Check if a queue can be sorted into another queue using a stack	53
Source	60
11 Check if an array is stack sortable	61
Source	62
12 Check if concatenation of two strings is balanced or not	63
Source	70
13 Check if two expressions with brackets are same	71
Source	74

14 Check if two trees are Mirror	75
Source	80
15 Check mirror in n-ary tree	81
Source	83
16 Computer Organization Stack based CPU Organization	84
Source	85
17 Construct BST from given preorder traversal Set 2	86
Source	91
18 Construct Binary Tree from String with bracket representation	92
Source	95
19 Convert Infix To Prefix Notation	96
Source	99
20 Count natural numbers whose all permutation are greater than that number	100
Source	102
21 Count subarrays where second highest lie before highest	103
Source	106
22 Create a customized data structure which evaluates functions in O(1)	107
Source	110
23 Decode a string recursively encoded as count followed by substring	111
Source	117
24 Delete array elements which are smaller than next or become smaller	118
Source	121
25 Delete consecutive same words in a sequence	122
Source	127
26 Delete middle element of a stack	128
Source	134
27 Design a stack that supports getMin() in O(1) time and O(1) extra space	135
Source	144
28 Design a stack to retrieve original elements and return the minimum element in O(1) time and O(1) space	145
Source	151
29 Design a stack with operations on middle element	152
Source	158

30 Design and Implement Special Stack Data Structure Added Space Optimized Version	159
Source	166
31 Evaluation of Prefix Expressions	167
Source	171
32 Expression Evaluation	172
Source	179
33 Expression contains redundant bracket or not	180
Source	183
34 Find if an expression has duplicate parenthesis or not	184
Source	186
35 Find index of closing bracket for a given opening bracket in an expression	187
Source	190
36 Find maximum depth of nested parenthesis in a string	191
Source	195
37 Find maximum difference between nearest left and right smaller elements	196
Source	202
38 Find maximum of minimum for every window size in a given array	203
Source	213
39 Find maximum sum possible equal sum of three stacks	214
Source	223
40 Find next Smaller of next Greater in an array	224
Source	229
41 Find the nearest smaller numbers on left side in an array	230
Source	237
42 Form minimum number from given sequence	238
Source	249
43 Growable array based stack	250
Source	253
44 How to create mergable stack?	254
Source	255
45 How to efficiently implement k stacks in a single array?	256
Source	262
46 How to implement stack using priority queue or heap?	263
Source	266

47 Identify and mark unmatched parenthesis in an expression	267
Source	269
48 Implement Stack and Queue using Deque	270
Source	276
49 Implement Stack using Queues	277
Source	283
50 Implement a stack using single queue	284
Source	288
51 Implement two stacks in an array	289
Source	296
52 Infix to Prefix conversion using two stacks	297
Source	304
53 Inorder Tree Traversal without Recursion	305
Source	315
54 Interleave the first half of the queue with second half	316
Source	318
55 Iterative Depth First Traversal of Graph	319
Source	328
56 Iterative Postorder Traversal Set 1 (Using Two Stacks)	329
Source	336
57 Iterative Postorder Traversal Set 2 (Using One Stack)	337
Source	346
58 Iterative Tower of Hanoi	347
Source	355
59 Iterative method to find ancestors of a given binary tree	356
Source	359
60 LIFO (Last-In-First-Out) approach in Programming	360
Source	363
61 Largest Rectangular Area in a Histogram Set 1	364
Source	368
62 Largest Rectangular Area in a Histogram Set 2	369
Source	373
63 Length of the longest valid substring	374
Source	379

64 Level order traversal in spiral form	380
Source	389
65 Level order traversal in spiral form Using one stack and one queue	390
Source	392
66 Level order traversal with direction change after every two levels	393
Source	400
67 Maximum length of rod for Q-th person	401
Source	404
68 Maximum product of indexes of next greater on left and right	405
Source	412
69 Maximum size rectangle binary sub-matrix with all 1s	413
Source	419
70 Maximum sum of smallest and second smallest in an array	420
Source	424
71 Merge Overlapping Intervals	425
Source	429
72 Merging and Sorting Two Unsorted Stacks	430
Source	436
73 Minimum number of bracket reversals needed to make an expression balanced	437
Source	441
74 Modify a binary tree to get preorder traversal using right pointers only	442
Source	447
75 Next Greater Element	448
Source	463
76 Next Greater Frequency Element	464
Source	467
77 Next Smaller Element	468
Source	479
78 Number of NGEs to the right	480
Source	483
79 Pattern Occurrences : Stack Implementation Java	484
Source	490
80 Postfix to Infix	491
Source	493

81 Postfix to Prefix Conversion	494
Source	496
82 Prefix to Infix Conversion	497
Source	499
83 Prefix to Postfix Conversion	500
Source	502
84 Preorder from Inorder and Postorder traversals	503
Source	508
85 Previous greater element	509
Source	516
86 Print Bracket Number	517
Source	524
87 Print Reverse a linked list using Stack	525
Source	528
88 Print ancestors of a given binary tree node without recursion	529
Source	538
89 Print next greater number of Q queries	539
Source	545
90 Program for Tower of Hanoi	546
Source	550
91 Queue using Stacks	551
Source	565
92 Range Queries for Longest Correct Bracket Subsequence	566
Source	571
93 Range Queries for Longest Correct Bracket Subsequence Set 2	572
Source	575
94 Rat in a Maze Backtracking using Stack	576
Source	581
95 Remove brackets from an algebraic string containing + and – operators	582
Source	584
96 Remove repeated digits in a given number	585
Source	590
97 Reverse a number using stack	591
Source	594

98 Reverse a stack using recursion	595
Source	605
99 Reverse a stack without using extra space in $O(n)$	606
Source	610
100 Reverse individual words	611
Source	614
101 Reversing a Queue	615
Source	618
102 Reversing the first K elements of a Queue	619
Source	623
103 Simplify the directory path (Unix like)	624
Source	627
104 Sort a stack using a temporary stack	628
Source	634
105 Sort a stack using recursion	635
Source	642
106 Sort string of characters using Stack	643
Source	645
107 Sorting array using Stacks	646
Source	651
108 Spaghetti Stack	652
Source	653
109 Stack Class in Java	654
Source	657
110 Stack Data Structure (Introduction and Program)	658
Source	669
111 Stack Permutations (Check if an array is stack permutation of other)	670
Source	673
112 Stack and Queue in Python using queue Module	674
Source	677
113 Stack Set 2 (Infix to Postfix)	678
Source	687
114 Stack Set 3 (Reverse a string using stack)	688
Source	696

115 Stack Set 4 (Evaluation of Postfix Expression)	697
Source	707
116 Sudo Placement[1.3] Playing with Stacks	708
Source	710
117 Sudo Placement[1.3] Stack Design	711
Source	713
118 The Celebrity Problem	714
Source	726
119 The Stock Span Problem	727
Source	738
120 Tracking current Maximum Element in a Stack	739
Source	742
121 Water drop problem	743
Source	745
122 ZigZag Tree Traversal	746
Source	752

Chapter 1

Alternate Odd and Even Nodes in a Singly Linked List

Alternate Odd and Even Nodes in a Singly Linked List - GeeksforGeeks

Given a singly linked list, rearrange the list so that even and odd nodes are alternate in the list.

There are two possible forms of this rearrangement. If the first data is odd, then the second node must be even. The third node must be odd and so on. Notice that another arrangement is possible where the first node is even, second odd, third even and so on.

Examples:

```
Input : 11 -> 20 -> 40 -> 55 -> 77 -> 80 -> NULL
Output : 11 -> 20 -> 55 -> 40 -> 77 -> 80 -> NULL
20, 40, 80 occur in even positions and 11, 55, 77
occur in odd positions.
```

```
Input : 10 -> 1 -> 2 -> 3 -> 5 -> 6 -> 7 -> 8 -> NULL
Output : 1 -> 10 -> 3 -> 2 -> 5 -> 6 -> 7 -> 8 -> NULL
1, 3, 5, 7 occur in odd positions and 10, 2, 6, 8 occur
at even positions in the list
```

Method 1 (Simple)

In this method, we create two stacks-Odd and Even. We traverse the list and when we encounter an even node in an odd position we push this node's address onto Even Stack. If we encounter an odd node in even position we push this node's address onto Odd Stack. After traversing the list, we simply pop the nodes at the top of the two stacks and exchange their data. We keep repeating this step until the stacks become empty.

Step 1: Create two stacks Odd and Even. These stacks will store the pointers to the nodes in the list

Step 2: Traverse list from start to end, using the variable current. Repeat following steps 3-4

Step 3: If current node is even and it occurs at an odd position, push this node's address to stack Even

Step 4: If current node is odd and it occurs at an even position, push this node's address to stack Odd.

[END OF TRAVERSAL]

Step 5: The size of both the stacks will be same. While both the stacks are not empty exchange the nodes at the top of the two stacks, and pop both nodes from their respective stacks.

Step 6: The List is now rearranged. STOP

C++

```
// CPP program to rearrange nodes
// as alternate odd even nodes in
// a Singly Linked List
#include <bits/stdc++.h>
using namespace std;

// Structure node
struct Node {
    int data;
    struct Node* next;
};

// A utility function to print
// linked list
void printList(struct Node* node)
{
    while (node != NULL) {
        cout << node->data << " ";
        node = node->next;
    }
    cout << endl;
}

// Function to create newNode
// in a linkedlist
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// Function to insert at beginning
```

```
Node* insertBeg(Node* head, int val)
{
    Node* temp = newNode(val);
    temp->next = head;
    head = temp;
    return head;
}

// Function to rearrange the
// odd and even nodes
void rearrangeOddEven(Node* head)
{
    stack<Node*> odd;
    stack<Node*> even;
    int i = 1;

    while (head != nullptr) {

        if (head->data % 2 != 0 && i % 2 == 0) {

            // Odd Value in Even Position
            // Add pointer to current node
            // in odd stack
            odd.push(head);
        }

        else if (head->data % 2 == 0 && i % 2 != 0) {

            // Even Value in Odd Position
            // Add pointer to current node
            // in even stack
            even.push(head);
        }

        head = head->next;
        i++;
    }

    while (!odd.empty() && !even.empty()) {

        // Swap Data at the top of two stacks
        swap(odd.top()->data, even.top()->data);
        odd.pop();
        even.pop();
    }
}

// Driver code
```

```
int main()
{
    Node* head = newNode(8);
    head = insertBeg(head, 7);
    head = insertBeg(head, 6);
    head = insertBeg(head, 5);
    head = insertBeg(head, 3);
    head = insertBeg(head, 2);
    head = insertBeg(head, 1);

    cout << "Linked List:" << endl;
    printList(head);
    rearrangeOddEven(head);

    cout << "Linked List after "
         << "Rearranging:" << endl;
    printList(head);

    return 0;
}
```

Output:

```
Linked List:
1 2 3 5 6 7 8
Linked List after Rearranging:
1 2 3 6 5 8 7
```

Time Complexity : $O(n)$

Auxiliary Space : $O(n)$

Method 2 (Efficient)

1. Segregate odd and even values in the list. After this, all odd values will occur together followed by all even values.
2. Split the list into two lists odd and even.
3. Merge the even list into odd list

REARRANGE (HEAD)

Step 1: Traverse the list using NODE TEMP.

 If TEMP is odd

 Add TEMP to the beginning of the List

 [END OF IF]

 [END OF TRAVERSAL]

Step 2: Set TEMP to 2nd element of LIST.

```
Step 3: Set PREV_TEMP to 1st element of List
Step 4: Traverse using node TEMP as long as an even
       node is not encountered.
       PREV_TEMP = TEMP, TEMP = TEMP->NEXT
       [END OF TRAVERSAL]
Step 5: Set EVEN to TEMP. Set PREV_TEMP->NEXT to NULL
Step 6: I = HEAD, J = EVEN
Step 7: Repeat while I != NULL and J != NULL
       Store next nodes of I and J in K and L
       K = I->NEXT, L = J->NEXT
       I->NEXT = J, J->NEXT = K, PTR = J
       I = K and J = L
       [END OF LOOP]
Step 8: if I == NULL
       PTR->NEXT = J
       [END of IF]
Step 8: Return HEAD.
Step 9: End
```

C++

```
// Cpp program to rearrange nodes
// as alternate odd even nodes in
// a Singly Linked List
#include <iostream>
#include <stack>
using namespace std;

// Structure node
struct Node {
    int data;
    struct Node* next;
};

// A utility function to print
// linked list
void printList(struct Node* node)
{
    while (node != NULL) {
        cout << node->data << " ";
        node = node->next;
    }
    cout << endl;
}

// Function to create newNode
// in a linkedlist
Node* newNode(int key)
```

```
{
    Node* temp = new Node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// Function to insert at beginning
Node* insertBeg(Node* head, int val)
{
    Node* temp = newNode(val);
    temp->next = head;
    head = temp;
    return head;
}

// Function to rearrange the
// odd and even nodes
void rearrange(Node** head)
{
    // Step 1: Segregate even and odd nodes
    // Step 2: Split odd and even lists
    // Step 3: Merge even list into odd list
    Node* even;
    Node *temp, *prev_temp;
    Node *i, *j, *k, *l, *ptr;

    // Step 1: Segregate Odd and Even Nodes
    temp = (*head)->next;
    prev_temp = *head;

    while (temp != nullptr) {

        // Backup next pointer of temp
        Node* x = temp->next;

        // If temp is odd move the node
        // to beginning of list
        if (temp->data % 2 != 0) {
            prev_temp->next = x;
            temp->next = (*head);
            (*head) = temp;
        }
        else {
            prev_temp = temp;
        }

        // Advance Temp Pointer
    }
}
```

```
    temp = x;
}

// Step 2
// Split the List into Odd and even
temp = (*head)->next;
prev_temp = (*head);

while (temp != nullptr && temp->data % 2 != 0) {
    prev_temp = temp;
    temp = temp->next;
}

even = temp;

// End the odd List (Make last node null)
prev_temp->next = nullptr;

// Step 3:
// Merge Even List into odd
i = *head;
j = even;

while (j != nullptr && i != nullptr) {

    // While both lists are not
    // exhausted Backup next
    // pointers of i and j
    k = i->next;
    l = j->next;

    i->next = j;
    j->next = k;

    // ptr points to the latest node added
    ptr = j;

    // Advance i and j pointers
    i = k;
    j = l;
}

if (i == nullptr) {

    // Odd list exhausts before even,
    // append remainder of even list to odd.
    ptr->next = j;
}
```

```
// The case where even list exhausts before
// odd list is automatically handled since we
// merge the even list into the odd list
}

// Driver Code
int main()
{
    Node* head = newNode(8);
    head = insertBeg(head, 7);
    head = insertBeg(head, 6);
    head = insertBeg(head, 3);
    head = insertBeg(head, 5);
    head = insertBeg(head, 1);
    head = insertBeg(head, 2);
    head = insertBeg(head, 10);

    cout << "Linked List:" << endl;
    printList(head);
    cout << "Rearranged List" << endl;
    rearrange(&head);
    printList(head);
}
```

Output:

```
Linked List:
10 2 1 5 3 6 7 8
Rearranged List
7 10 3 2 5 6 1 8
```

Time Complexity : $O(n)$
Auxiliary Space : $O(1)$

Source

<https://www.geeksforgeeks.org/alternate-odd-even-nodes-singly-linked-list/>

Chapter 2

Arithmetic Expression Evaluation

Arithmetic Expression Evaluation - GeeksforGeeks

The stack organization is very effective in evaluating arithmetic expressions. Expressions are usually represented in what is known as **Infix notation**, in which each operator is written between two operands (i.e., $A + B$). With this notation, we must distinguish between $(A + B) * C$ and $A + (B * C)$ by using either parentheses or some operator-precedence convention. Thus, the order of operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

1. **Polish notation (prefix notation)** –

It refers to the notation in which the operator is placed before its two operands. Here no parentheses are required, i.e.,

$+AB$

2. **Reverse Polish notation (postfix notation)** –

It refers to the analogous notation in which the operator is placed after its two operands. Again, no parentheses are required in Reverse Polish notation, i.e.,

$AB+$

Stack organized computers are better suited for post-fix notation than the traditional infix notation. Thus the infix notation must be converted to the post-fix notation. The conversion from infix notation to post-fix notation must take into consideration the operational hierarchy.

There are 3 levels of precedence for 5 binary operators as given below:

Highest: Exponentiation (^)

Next highest: Multiplication (*) and division (/)

Lowest: Addition (+) and Subtraction (-)

For example –

Infix notation: $(A-B) * [C / (D+E) + F]$

Post-fix notation: $AB- CDE +/F +*$

Here, we first perform the arithmetic inside the parentheses $(A-B)$ and $(D+E)$. The division of $C/(D+E)$ must be done prior to the addition with F . After that multiply the two terms inside the parentheses and bracket.

Now we need to calculate the value of these arithmetic operations by using stack.

The procedure for getting the result is:

1. Convert the expression in Reverse Polish notation(post-fix notation).
2. Push the operands into the stack in the order they appear.
3. When any operator encounter then pop two topmost operands for executing the operation.
4. After execution push the result obtained into the stack.
5. After the complete execution of expression the final result remains on the top of the stack.

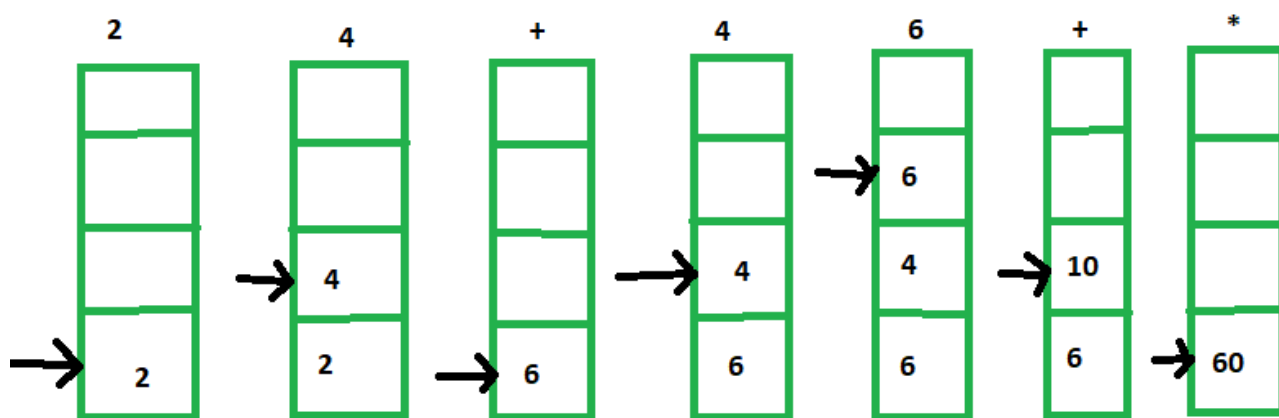
For example –

Infix notation: $(2+4) * (4+6)$

Post-fix notation: $2\ 4\ +\ 4\ 6\ +\ *$

Result: 60

The stack operations for this expression evaluation is shown below:



Stack operations to evaluate $(2+4)*(4+6)$

Source

<https://www.geeksforgeeks.org/arithmetic-expression-evaluation/>

Chapter 3

Balanced expression with replacement

Balanced expression with replacement - GeeksforGeeks

Given a string that contains only the following => '{', '}', '(', ')', '[', ']', 'X'. At some places there is 'X' in place of any bracket. Determine whether by replacing all 'X's with appropriate bracket, is it possible to make a valid bracket sequence.

Prerequisite: [Balanced Parenthesis Expression](#)

Examples:

Input : S = "{(X[X])}"
Output : Balanced
The balanced expression after
replacing X with suitable bracket is:
{([[]])}.

Input : [{X}(X)]
Output : Not balanced
No substitution of X with any bracket
results in a balanced expression.

Approach: We have discussed a solution on verifying whether given [parenthesis expression is balanced or not](#).

Following the same approach described in the article, a stack data structure is used for verifying whether given expression is balanced or not. For each type of character in string, the operations to be performed on stack are:

1. '{' or '(' or '[' : When current element of string is an opening bracket, push the element in stack.

2. '}' or ']' or ')' : When current element of string is a closing bracket, pop the top element of the stack and check if it is a matching opening bracket for the closing bracket or not. If it is matching, then move to next element of string. If it is not, then current string is not balanced. It is also possible that the element popped from stack is 'X'. In that case 'X' is a matching opening bracket because it is pushed in stack only when it is assumed to be an opening bracket as described in next step.
3. 'X' : When current element is X then it can either be a starting bracket or a closing bracket. First assume that it is a starting bracket and recursively call for next element by pushing X in stack. If the result of recursion is false then X is a closing bracket which matches the bracket at top of the stack (If stack is non-empty). So pop the top element and recursively call for next element. If the result of recursion is again false, then the expression is not balanced.

Also check for the case when stack is empty and current element is a closing bracket. In that case, the expression is not balanced.

Implementation:

C++

```
// CPP program to determine whether given
// expression is balanced/ parenthesis
// expression or not.
#include <bits/stdc++.h>
using namespace std;

// Function to check if two brackets are matching
// or not.
int isMatching(char a, char b)
{
    if ((a == '{' && b == '}') || (a == '[' && b == ']')
        || (a == '(' && b == ')') || a == 'X')
        return 1;
    return 0;
}

// Recursive function to check if given expression
// is balanced or not.
int isBalanced(string s, stack<char> ele, int ind)
{
    // Base case.
    // If the string is balanced then all the opening
    // brackets had been popped and stack should be
    // empty after string is traversed completely.
    if (ind == s.length())
        return ele.empty();
}
```

```
// variable to store element at the top of the stack.
char topEle;

// variable to store result of recursive call.
int res;

// Case 1: When current element is an opening bracket
// then push that element in the stack.
if (s[ind] == '{' || s[ind] == '(' || s[ind] == '[') {
    ele.push(s[ind]);
    return isBalanced(s, ele, ind + 1);
}

// Case 2: When current element is a closing bracket
// then check for matching bracket at the top of the
// stack.
else if (s[ind] == '}' || s[ind] == ')' || s[ind] == ']') {

    // If stack is empty then there is no matching opening
    // bracket for current closing bracket and the
    // expression is not balanced.
    if (ele.empty())
        return 0;

    topEle = ele.top();
    ele.pop();

    // Check bracket is matching or not.
    if (!isMatching(topEle, s[ind]))
        return 0;

    return isBalanced(s, ele, ind + 1);
}

// Case 3: If current element is 'X' then check
// for both the cases when 'X' could be opening
// or closing bracket.
else if (s[ind] == 'X') {
    stack<char> tmp = ele;
    tmp.push(s[ind]);
    res = isBalanced(s, tmp, ind + 1);
    if (res)
        return 1;
    if (ele.empty())
        return 0;
    ele.pop();
    return isBalanced(s, ele, ind + 1);
}
```

```
}

int main()
{
    string s = "{(X}[]";
    stack<char> ele;
    if (isBalanced(s, ele, 0))
        cout << "Balanced";
    else
        cout << "Not Balanced";
    return 0;
}
```

C#

```
// C# program to determine
// whether given expression
// is balanced/ parenthesis
// expression or not.
using System;
using System.Collections.Generic;

class GFG
{
    // Function to check if two
    // brackets are matching or not.
    static int isMatching(char a,
                          char b)
    {
        if ((a == '{' && b == '}') ||
            (a == '[' && b == ']') ||
            (a == '(' && b == ')') || a == 'X')
            return 1;
        return 0;
    }

    // Recursive function to
    // check if given expression
    // is balanced or not.
    static int isBalanced(string s,
                          Stack<char> ele,
                          int ind)
    {
        // Base case.
        // If the string is balanced
        // then all the opening brackets
        // had been popped and stack
```

```
// should be empty after string
// is traversed completely.
if (ind == s.Length)
{
    if (ele.Count == 0)
        return 1;
    else
        return 0;
}

// variable to store element
// at the top of the stack.
char topEle;

// variable to store result
// of recursive call.
int res;

// Case 1: When current element
// is an opening bracket
// then push that element
// in the stack.
if (s[ind] == '{' ||
    s[ind] == '(' ||
    s[ind] == '[')
{
    ele.Push(s[ind]);
    return isBalanced(s, ele, ind + 1);
}

// Case 2: When current element
// is a closing bracket then
// check for matching bracket
// at the top of the stack.
else if (s[ind] == '}' ||
        s[ind] == ')' ||
        s[ind] == ']')
{
    // If stack is empty then there
    // is no matching opening bracket
    // for current closing bracket and
    // the expression is not balanced.
    if (ele.Count == 0)
        return 0;

    topEle = ele.Peek();
    ele.Pop();
}
```



```
        // Check bracket is
        // matching or not.
        if (isMatching(topEle, s[ind]) == 0)
            return 0;

        return isBalanced(s, ele, ind + 1);
    }

    // Case 3: If current element
    // is 'X' then check for both
    // the cases when 'X' could be
    // opening or closing bracket.
    else if (s[ind] == 'X')
    {
        Stack<char> tmp = ele;
        tmp.Push(s[ind]);
        res = isBalanced(s, tmp, ind + 1);
        if (res == 1)
            return 1;
        if (ele.Count == 0)
            return 0;
        ele.Pop();
        return isBalanced(s, ele, ind + 1);
    }
    return 1;
}

// Driver Code
static void Main()
{
    string s = "{(X)}[]";
    Stack<char> ele = new Stack<char>();

    if (isBalanced(s, ele, 0) != 0)
        Console.Write("Balanced");
    else
        Console.Write("Not Balanced");
}

// This code is contributed by
// Manish Shaw(manishshaw1)
```

Output:

Balanced

Time Complexity: $O((2^n) * n)$

Auxiliary Space: $O(N)$

Improved By : [manishshaw1](#)

Source

<https://www.geeksforgeeks.org/balanced-expression-replacement/>

Chapter 4

Bank Of America (BA Continuum India Pvt. Ltd.) Campus Recruitment

Bank Of America (BA Continuum India Pvt. Ltd.) Campus Recruitment - GeeksforGeeks
Approved Offer.

Bank Of America has visited our college for on campus recruitment . The recruitment consisted of 4 Rounds in total.

The recruitment was for BA Continuum India Pvt Ltd. the technical field of BOA

Round 1:

This round was a general aptitude test, English proficiency, quantitative analysis and the logical questions. This was time specific for each and every section. Try practicing from GFG, indiabix and careerride youtube videos.

Round 2: TECHNICAL 1

This was a face to face technical interview. I was asked to submit my resume and then was asked to wait until my name was called out. The interview went around 30-40 minutes.

The interviewer started with a very basic aptitude problem(dices probability) and then he asked me a puzzle of water jug problem. Then he asked me to explain my projects in the resume and about the experience I had in the previous organization.

After that he went forward with the technical questions consisting of CODING problems .

1. What is recursion? Find the length of a linked list using recursion
This was to test the basics of the datastructure
2. Balanced Parentheses problem
3. Find the position of the first unbalanced parentheses?

Round 3: TECHNICAL 2

It was another technical round. This round was based on advanced coding.

1. Base conversion. Given a number in 10 format convert it into base of 6?
2. Matrix generation.
3. Backtracking question

The interviewers above solely focused on the approach or the pseudo code and helped if we got stuck somewhere.

Round 4: HR ROUND

Here the HR was very friendly and asked about whether you are able to relocate if asked?

At last I was given a feedback by the interviewers as strong logical and coding skills . I received the offer letter on that day itself at night.

Position: Senior Tech Associate, Bank Of America

Source

<https://www.geeksforgeeks.org/bank-of-america-ba-continuum-india-pvt-ltd-campus-recruitment/>

Chapter 5

Bubble sort using two Stacks

Bubble sort using two Stacks - GeeksforGeeks

Prerequisite : [Bubble Sort](#)

Write a function that sort an array of integers using stacks and also uses bubble sort paradigm.

Algorithm:

1. Push all elements of array in 1st stack
 2. Run a loop for 'n' times(n is size of array) having the following :
 - 2.a. Keep on pushing elements in the 2nd stack till the top of second stack is smaller than element being pushed from 1st stack.
 - 2.b. If the element being pushed is smaller than top of 2nd stack then swap them (as in bubble sort)
- *Do above steps alternatively

TRICKY STEP: Once a stack is empty, then the top of the next stack will be the largest number so keep it at its position in array i.e arr[len-1-i] and then pop it from that stack.

```
// Java program for bubble sort
// using stack

import java.util.Arrays;
```

```
import java.util.Stack;

public class Test
{
    // Method for bubble sort using Stack
    static void bubbleSortStack(int arr[], int n)
    {
        Stack<Integer> s1 = new Stack<>();

        // Push all elements of array in 1st stack
        for (int num : arr)
            s1.push(num);

        Stack<Integer> s2 = new Stack<>();

        for (int i = 0; i < n; i++)
        {
            // alternatively
            if (i % 2 == 0)
            {
                while (!s1.isEmpty())
                {
                    int t = s1.pop();

                    if (s2.isEmpty())
                        s2.push(t);
                    else
                    {
                        if (s2.peek() > t)
                        {
                            // swapping
                            int temp = s2.pop();
                            s2.push(t);
                            s2.push(temp);
                        }
                        else
                        {
                            s2.push(t);
                        }
                    }
                }

                // tricky step
                arr[n-1-i] = s2.pop();
            }
            else
            {
                while (!s2.isEmpty())
```

```
        {
            int t = s2.pop();

            if (s1.isEmpty())
                s1.push(t);

            else
            {
                if (s1.peek() > t)
                {
                    // swapping
                    int temp = s1.pop();

                    s1.push(t);
                    s1.push(temp);
                }
                else
                    s1.push(t);
            }
        }

        // tricky step
        arr[n-1-i] = s1.pop();
    }
}
System.out.println(Arrays.toString(arr));
}

// Driver Method
public static void main(String[] args)
{
    int arr[] = {15, 12, 44, 2, 5, 10};
    bubbleSortStack(arr, arr.length);
}
}
```

Output:

[2, 5, 10, 12, 15, 44]

Source

<https://www.geeksforgeeks.org/bubble-sort-using-two-stacks/>

Chapter 6

Check if stack elements are pairwise consecutive

Check if stack elements are pairwise consecutive - GeeksforGeeks

Given a stack of integers, write a function `pairWiseConsecutive()` that checks whether numbers in the stack are pairwise consecutive or not. The pairs can be increasing or decreasing, and if the stack has an odd number of elements, the element at the top is left out of a pair. The function should retain the original stack content.

Only following standard operations are allowed on stack.

- `push(X)`: Enter a element X on top of stack.
- `pop()`: Removes top element of the stack.
- `empty()`: To check if stack is empty.

Examples:

Input : `stack = [4, 5, -2, -3, 11, 10, 5, 6, 20]`

Output : Yes

Each of the pairs (4, 5), (-2, -3), (11, 10) and (5, 6) consists of consecutive numbers.

Input : `stack = [4, 6, 6, 7, 4, 3]`

Output : No

(4, 6) are not consecutive.

The idea is to use another stack.

1. Create an auxiliary stack **aux**.
2. Transfer contents of given stack to **aux**.

3. Traverse aux. While traversing fetch top two elements and check if they are consecutive or not. After checking put these elements back to original stack.

```
// C++ program to check if successive
// pair of numbers in the stack are
// consecutive or not
#include <bits/stdc++.h>
using namespace std;

// Function to check if elements are
// pairwise consecutive in stack
bool pairWiseConsecutive(stack<int> s)
{
    // Transfer elements of s to aux.
    stack<int> aux;
    while (!s.empty()) {
        aux.push(s.top());
        s.pop();
    }

    // Traverse aux and see if
    // elements are pairwise
    // consecutive or not. We also
    // need to make sure that original
    // content is retained.
    bool result = true;
    while (aux.empty() > 1) {

        // Fetch current top two
        // elements of aux and check
        // if they are consecutive.
        int x = aux.top();
        aux.pop();
        int y = aux.top();
        aux.pop();
        if (abs(x - y) != 1)
            result = false;

        // Push the elements to original
        // stack.
        s.push(x);
        s.push(y);
    }

    if (aux.size() == 1)
        s.push(aux.top());

    return result;
}
```

```
}

// Driver program
int main()
{
    stack<int> s;
    s.push(4);
    s.push(5);
    s.push(-2);
    s.push(-3);
    s.push(11);
    s.push(10);
    s.push(5);
    s.push(6);
    s.push(20);

    if (pairWiseConsecutive(s))
        cout << "Yes" << endl;
    else
        cout << "No" << endl;

    cout << "Stack content (from top)"
         << " after function call\n";
    while (s.empty() == false)
    {
        cout << s.top() << " ";
        s.pop();
    }

    return 0;
}
```

Output:

```
Yes
Stack content (from top) after function call
20 6 5 10 11 -3 -2 5 4
```

Time complexity: $O(n)$.
Auxiliary Space : $O(n)$.

Source

<https://www.geeksforgeeks.org/check-if-stack-elements-are-pairwise-consecutive/>

Chapter 7

Check for balanced parentheses in an expression

Check for balanced parentheses in an expression - GeeksforGeeks

Given an expression string `exp` , write a program to examine whether the pairs and the orders of “{“,”}”,“(“,”)”,”[“,”]” are correct in `exp`. For example, the program should print true for `exp` = “[()]{}{[(())]} ” and false for `exp` = “[()]”

Checking for balanced parentheses is one of the most important task of a compiler.

```
int main(){  
    for ( int i=0; i < 10; i++)  
    {  
        //some code  
    }  
}  
} ← Compiler generates error
```

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”

C++

```
// CPP program to check for balanced parenthesis.
```

```
#include<bits/stdc++.h>
using namespace std;

// function to check if paranthesis are balanced
bool areParanthesisBalanced(string expr)
{
    stack<char> s;
    char x;

    // Traversing the Expression
    for (int i=0; i<expr.length(); i++)
    {
        if (expr[i]=='('||expr[i]=='['||expr[i]=='{')
        {
            // Push the element in the stack
            s.push(expr[i]);
            continue;
        }

        // IF current current character is not opening
        // bracket, then it must be closing. So stack
        // cannot be empty at this point.
        if (s.empty())
            return false;

        switch (expr[i])
        {
            case ')':

                // Store the top element in a
                x = s.top();
                s.pop();
                if (x=='{' || x=='[')
                    return false;
                break;

            case '}':

                // Store the top element in b
                x = s.top();
                s.pop();
                if (x=='(' || x=='[')
                    return false;
                break;

            case ']':

                // Store the top element in c
```

```
        x = s.top();
        s.pop();
        if (x == '(' || x == '{')
            return false;
        break;
    }
}

// Check Empty Stack
return (s.empty());
}

// Driver program to test above function
int main()
{
    string expr = "{()}[]";

    if (areParanthesisBalanced(expr))
        cout << "Balanced";
    else
        cout << "Not Balanced";
    return 0;
}
```

C

```
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* structure of a stack node */
struct sNode
{
    char data;
    struct sNode *next;
};

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data);

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref);

/* Returns 1 if character1 and character2 are matching left
and right Parenthesis */
bool isMatchingPair(char character1, char character2)
{
    if (character1 == '(' && character2 == ')')
```

```
        return 1;
    else if (character1 == '{' && character2 == '}')
        return 1;
    else if (character1 == '[' && character2 == ']')
        return 1;
    else
        return 0;
}

/*Return 1 if expression has balanced Parenthesis */
bool areParenthesisBalanced(char exp[])
{
    int i = 0;

    /* Declare an empty character stack */
    struct sNode *stack = NULL;

    /* Traverse the given expression to check matching parenthesis */
    while (exp[i])
    {
        /*If the exp[i] is a starting parenthesis then push it*/
        if (exp[i] == '{' || exp[i] == '(' || exp[i] == '[')
            push(&stack, exp[i]);

        /* If exp[i] is an ending parenthesis then pop from stack and
           check if the popped parenthesis is a matching pair*/
        if (exp[i] == '}' || exp[i] == ')' || exp[i] == ']')
        {
            /*If we see an ending parenthesis without a pair then return false*/
            if (stack == NULL)
                return 0;

            /* Pop the top element from stack, if it is not a pair
               parenthesis of character then there is a mismatch.
               This happens for expressions like {({}) */
            else if ( !isMatchingPair(pop(&stack), exp[i]) )
                return 0;
        }
        i++;
    }

    /* If there is something left in expression then there is a starting
       parenthesis without a closing parenthesis */
    if (stack == NULL)
        return 1; /*balanced*/
    else
        return 0; /*not balanced*/
}
```

```
}

/* UTILITY FUNCTIONS */
/*driver program to test above functions*/
int main()
{
    char exp[100] = "{}[]";
    if (areParenthesisBalanced(exp))
        printf("Balanced \n");
    else
        printf("Not Balanced \n");
    return 0;
}

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data)
{
    /* allocate node */
    struct sNode* new_node =
        (struct sNode*) malloc(sizeof(struct sNode));

    if (new_node == NULL)
    {
        printf("Stack overflow n");
        getchar();
        exit(0);
    }

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*top_ref);

    /* move the head to point to the new node */
    (*top_ref) = new_node;
}

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref)
{
    char res;
    struct sNode *top;

    /*If stack is empty then error */
    if (*top_ref == NULL)
    {
        printf("Stack overflow n");
    }
}
```

```
        getchar();
        exit(0);
    }
    else
    {
        top = *top_ref;
        res = top->data;
        *top_ref = top->next;
        free(top);
        return res;
    }
}
```

Java

```
// Java program for checking
// balanced Parenthesis

public class BalancedParan
{
    static class stack
    {
        int top=-1;
        char items[] = new char[100];

        void push(char x)
        {
            if (top == 99)
            {
                System.out.println("Stack full");
            }
            else
            {
                items[++top] = x;
            }
        }

        char pop()
        {
            if (top == -1)
            {
                System.out.println("Underflow error");
                return '\0';
            }
            else
            {
                char element = items[top];
                top--;
            }
        }
    }
}
```



```

        return element;
    }
}

boolean isEmpty()
{
    return (top == -1) ? true : false;
}

/* Returns true if character1 and character2
   are matching left and right Parenthesis */
static boolean isMatchingPair(char character1, char character2)
{
    if (character1 == '(' && character2 == ')')
        return true;
    else if (character1 == '{' && character2 == '}')
        return true;
    else if (character1 == '[' && character2 == ']')
        return true;
    else
        return false;
}

/* Return true if expression has balanced
   Parenthesis */
static boolean areParenthesisBalanced(char exp[])
{
    /* Declare an empty character stack */
    stack st=new stack();

    /* Traverse the given expression to
       check matching parenthesis */
    for(int i=0;i<exp.length;i++)
    {

        /*If the exp[i] is a starting
           parenthesis then push it*/
        if (exp[i] == '{' || exp[i] == '(' || exp[i] == '[')
            st.push(exp[i]);

        /* If exp[i] is an ending parenthesis
           then pop from stack and check if the
           popped parenthesis is a matching pair*/
        if (exp[i] == '}' || exp[i] == ')' || exp[i] == ']')
        {

            /* If we see an ending parenthesis without

```

```

        a pair then return false*/
    if (st.isEmpty())
    {
        return false;
    }

    /* Pop the top element from stack, if
    it is not a pair parenthesis of character
    then there is a mismatch. This happens for
    expressions like {(}) */
    else if ( !isMatchingPair(st.pop(), exp[i]) )
    {
        return false;
    }
}

}

/* If there is something left in expression
then there is a starting parenthesis without
a closing parenthesis */

if (st.isEmpty())
    return true; /*balanced*/
else
{
    /*not balanced*/
    return false;
}

}

/* UTILITY FUNCTIONS */
/*driver program to test above functions*/
public static void main(String[] args)
{
    char exp[] = {'{','(','(',')','}', '[',']'};
    if (areParenthesisBalanced(exp))
        System.out.println("Balanced ");
    else
        System.out.println("Not Balanced ");
}

}

```

Output:
Balanced

Time Complexity: $O(n)$

Auxiliary Space: $O(n)$ for stack.

Source

<https://www.geeksforgeeks.org/check-for-balanced-parentheses-in-an-expression/>

Chapter 8

Check for balanced parenthesis without using stack

Check for balanced parenthesis without using stack - GeeksforGeeks

Given an expression string exp, write a program to examine whether the pairs and the orders of “{, }”, “(,)”, “[,]” are correct in exp.

Examples:

Input : exp = "[()]{}{[()()]()}"
Output : true

Input : exp = "[()]"
Output : false

We have discussed a [stack based solution](#). Here we are not allowed to use stack. Looks like this problem cannot be solved without extra space (please see comments at the end). We use [recursion](#) to solve the problem.

Below is the implementation of above algorithm:

```
// CPP program to check if parenthesis are
// balanced or not in an expression.
#include <bits/stdc++.h>
using namespace std;

char findClosing(char c)
{
    if (c == '(')
        return ')';
    if (c == '{')
```

```
        return '}';
    if (c == '[')
        return ']';
    return -1;
}

// function to check if parenthesis are
// balanced.
bool check(char expr[], int n)
{
    // Base cases
    if (n == 0)
        return true;
    if (n == 1)
        return false;
    if (expr[0] == ')' || expr[0] == '}' || expr[0] == ']')
        return false;

    // Search for closing bracket for first
    // opening bracket.
    char closing = findClosing(expr[0]);

    // count is used to handle cases like
    // "((()))". We basically need to
    // consider matching closing bracket.
    int i, count = 0;
    for (i = 1; i < n; i++) {
        if (expr[i] == expr[0])
            count++;
        if (expr[i] == closing) {
            if (count == 0)
                break;
            count--;
        }
    }

    // If we did not find a closing
    // bracket
    if (i == n)
        return false;

    // If closing bracket was next
    // to open
    if (i == 1)
        return check(expr + 2, n - 2);

    // If closing bracket was somewhere
    // in middle.
```

```
        return check(expr + 1, i - 1) && check(expr + i + 1, n - i - 1);
    }

    // Driver program to test above function
    int main()
    {
        char expr[] = "[()]";
        int n = strlen(expr);
        if (check(expr, n))
            cout << "Balanced";
        else
            cout << "Not Balanced";
        return 0;
    }
```

Output:

Not Balanced

The above solution is very inefficient compared to stack based solution. This seems to only useful for recursion practice problems.

Source

<https://www.geeksforgeeks.org/check-for-balanced-parenthesis-without-using-stack/>

Chapter 9

Check if a given array can represent Preorder Traversal of Binary Search Tree

Check if a given array can represent Preorder Traversal of Binary Search Tree - Geeks-forGeeks

Given an array of numbers, return true if given array can represent preorder traversal of a Binary Search Tree, else return false. Expected time complexity is $O(n)$.

Examples:

Input: `pre[] = {2, 4, 3}`

Output: `true`

Given array can represent preorder traversal of below tree

2

4

/

3

Input: `pre[] = {2, 4, 1}`

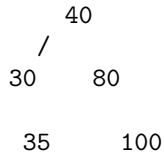
Output: `false`

Given array cannot represent preorder traversal of a Binary Search Tree.

Input: `pre[] = {40, 30, 35, 80, 100}`

Output: `true`

Given array can represent preorder traversal of below tree



Input: pre[] = {40, 30, 35, 20, 80, 100}
Output: false
Given array cannot represent preorder traversal
of a Binary Search Tree.

A **Simple Solution** is to do following for every node pre[i] starting from first one.

- 1) Find the first greater value on right side of current node.
Let the index of this node be j. Return true if following conditions hold. Else return false
 - (i) All values after the above found greater value are greater than current node.
 - (ii) Recursive calls for the subarrays pre[i+1..j-1] and pre[j+1..n-1] also return true.

Time Complexity of the above solution is $O(n^2)$

An **Efficient Solution** can solve this problem in $O(n)$ time. The idea is to use a stack. This problem is similar to [Next \(or closest\) Greater Element problem](#). Here we find next greater element and after finding next greater, if we find a smaller element, then return false.

- 1) Create an empty stack.
- 2) Initialize root as INT_MIN.
- 3) Do following for every element pre[i]
 - a) If pre[i] is smaller than current root, return false.
 - b) Keep removing elements from stack while pre[i] is greater than stack top. Make the last removed item as new root (to be compared next).
At this point, pre[i] is greater than the removed root (That is why if we see a smaller element in step a), we return false)
 - c) push pre[i] to stack (All elements in stack are in decreasing order)

Below is implementation of above idea.

C++


```
// C++ program for an efficient solution to check if
// a given array can represent Preorder traversal of
// a Binary Search Tree
#include<bits/stdc++.h>
using namespace std;

bool canRepresentBST(int pre[], int n)
{
    // Create an empty stack
    stack<int> s;

    // Initialize current root as minimum possible
    // value
    int root = INT_MIN;

    // Traverse given array
    for (int i=0; i<n; i++)
    {
        // If we find a node who is on right side
        // and smaller than root, return false
        if (pre[i] < root)
            return false;

        // If pre[i] is in right subtree of stack top,
        // Keep removing items smaller than pre[i]
        // and make the last removed item as new
        // root.
        while (!s.empty() && s.top()<pre[i])
        {
            root = s.top();
            s.pop();
        }

        // At this point either stack is empty or
        // pre[i] is smaller than root, push pre[i]
        s.push(pre[i]);
    }
    return true;
}

// Driver program
int main()
{
    int pre1[] = {40, 30, 35, 80, 100};
    int n = sizeof(pre1)/sizeof(pre1[0]);
    canRepresentBST(pre1, n)? cout << "truen":
                             cout << "falsen";
}
```

```
int pre2[] = {40, 30, 35, 20, 80, 100};
n = sizeof(pre2)/sizeof(pre2[0]);
canRepresentBST(pre2, n)? cout << "truen":
                        cout << "falsen";

return 0;
}
```

Java

```
// Java program for an efficient solution to check if
// a given array can represent Preorder traversal of
// a Binary Search Tree
import java.util.Stack;

class BinarySearchTree {

    boolean canRepresentBST(int pre[], int n) {
        // Create an empty stack
        Stack<Integer> s = new Stack<Integer>();

        // Initialize current root as minimum possible
        // value
        int root = Integer.MIN_VALUE;

        // Traverse given array
        for (int i = 0; i < n; i++) {
            // If we find a node who is on right side
            // and smaller than root, return false
            if (pre[i] < root) {
                return false;
            }

            // If pre[i] is in right subtree of stack top,
            // Keep removing items smaller than pre[i]
            // and make the last removed item as new
            // root.
            while (!s.empty() && s.peek() < pre[i]) {
                root = s.peek();
                s.pop();
            }

            // At this point either stack is empty or
            // pre[i] is smaller than root, push pre[i]
            s.push(pre[i]);
        }
        return true;
    }
}
```

```
public static void main(String args[]) {
    BinarySearchTree bst = new BinarySearchTree();
    int[] pre1 = new int[]{40, 30, 35, 80, 100};
    int n = pre1.length;
    if (bst.canRepresentBST(pre1, n) == true) {
        System.out.println("true");
    } else {
        System.out.println("false");
    }
    int[] pre2 = new int[]{40, 30, 35, 20, 80, 100};
    int n1 = pre2.length;
    if (bst.canRepresentBST(pre2, n) == true) {
        System.out.println("true");
    } else {
        System.out.println("false");
    }
}
```

//This code is contributed by Mayank Jaiswal

Python

```
# Python program for an efficient solution to check if
# a given array can represent Preorder traversal of
# a Binary Search Tree
```

```
INT_MIN = -2**32
```

```
def canRepresentBST(pre):
```

```
    # Create an empty stack
    s = []
```

```
    # Initialize current root as minimum possible value
    root = INT_MIN
```

```
    # Traverse given array
    for value in pre:
        #NOTE:value is equal to pre[i] according to the
        #given algo
```

```
        # If we find a node who is on the right side
        # and smaller than root, return False
        if value < root :
            return False
```

```
# If value(pre[i]) is in right subtree of stack top,
# Keep removing items smaller than value
# and make the last removed items as new root
while(len(s) > 0 and s[-1] < value) :
    root = s.pop()

# At this point either stack is empty or value
# is smaller than root, push value
s.append(value)

return True

# Driver Program
pre1 = [40 , 30 , 35 , 80 , 100]
print "true" if canRepresentBST(pre1) == True else "false"
pre2 = [40 , 30 , 35 , 20 , 80 , 100]
print "true" if canRepresentBST(pre2) == True else "false"

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
true
false
```

This article is contributed by [Romil Punetha](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/check-if-a-given-array-can-represent-preorder-traversal-of-binary-search-tree/>

Chapter 10

Check if a queue can be sorted into another queue using a stack

Check if a queue can be sorted into another queue using a stack - GeeksforGeeks

Given a Queue consisting of first n natural numbers (in random order). The task is to check whether the given Queue elements can be arranged in increasing order in another Queue using a stack. The operation allowed are:

1. Push and pop elements from the stack
2. Pop (Or enqueue) from the given Queue.
3. Push (Or Dequeue) in the another Queue.

Examples :

Input : Queue[] = { 5, 1, 2, 3, 4 }

Output : Yes

Pop the first element of the given Queue i.e 5.

Push 5 into the stack.

Now, pop all the elements of the given Queue and push them to second Queue.

Now, pop element 5 in the stack and push it to the second Queue.

Input : Queue[] = { 5, 1, 2, 6, 3, 4 }

Output : No

Push 5 to stack.

Pop 1, 2 from given Queue and push it to another Queue.

Pop 6 from given Queue and push to stack.

Pop 3, 4 from given Queue and push to second Queue.

Now, from using any of above operation, we cannot push 5 into the second Queue because it is below the 6 in the stack.

Observe, second Queue (which will contain the sorted element) takes inputs (or enqueue elements) either from given Queue or Stack. So, next expected (which will initially be 1)

element must be present as a front element of given Queue or top element of the Stack. So, simply simulate the process for the second Queue by initializing the expected element as 1. And check if we can get expected element from the front of the given Queue or from the top of the Stack. If we cannot take it from the either of them then pop the front element of given Queue and push it in the Stack.

Also, observe, that the stack must also be sorted at each instance i.e the element at the top of the stack must be smallest in the stack. For eg. let $x > y$, then x will always be expected before y . So, x cannot be pushed before y in the stack. Therefore, we cannot push element with the higher value on the top of the element having lesser value.

Algorithm:

1. Initialize the expected_element = 1
2. Check if either front element of given Queue or top element of the stack have expected_element
 -a) If yes, increment expected_element by 1, repeat step 2.
 -b) Else, pop front of Queue and push it to the stack. If the popped element is greater than top of the Stack, return "No".

Below is the implementation of this approach:

C++

```
// CPP Program to check if a queue of first
// n natural number can be sorted using a stack
#include <bits/stdc++.h>
using namespace std;

// Function to check if given queue element
// can be sorted into another queue using a
// stack.
bool checkSorted(int n, queue<int>& q)
{
    stack<int> st;
    int expected = 1;
    int fnt;

    // while given Queue is not empty.
    while (!q.empty()) {
        fnt = q.front();
        q.pop();

        // if front element is the expected element
        if (fnt == expected)
            expected++;

        else {
            // if stack is empty, push the element
            if (st.empty()) {
```

```
        st.push(fnt);
    }

    // if top element is less than element which
    // need to be pushed, then return false.
    else if (!st.empty() && st.top() < fnt) {
        return false;
    }

    // else push into the stack.
    else
        st.push(fnt);
}

// while expected element are coming from
// stack, pop them out.
while (!st.empty() && st.top() == expected) {
    st.pop();
    expected++;
}

// if the final expected element value is equal
// to initial Queue size and the stack is empty.
if (expected - 1 == n && st.empty())
    return true;

return false;
}

// Driven Program
int main()
{
    queue<int> q;
    q.push(5);
    q.push(1);
    q.push(2);
    q.push(3);
    q.push(4);

    int n = q.size();

    (checkSorted(n, q) ? (cout << "Yes") :
     (cout << "No"));

    return 0;
}
```

Java

```
// Java Program to check if a queue
// of first n natural number can
// be sorted using a stack
import java.io.*;
import java.util.*;

class GFG
{
    static Queue<Integer> q =
        new LinkedList<Integer>();

    // Function to check if given
    // queue element can be sorted
    // into another queue using a stack.
    static boolean checkSorted(int n)
    {
        Stack<Integer> st =
            new Stack<Integer>();
        int expected = 1;
        int fnt;

        // while given Queue
        // is not empty.
        while (q.size() != 0)
        {
            fnt = q.peek();
            q.poll();

            // if front element is
            // the expected element
            if (fnt == expected)
                expected++;

            else
            {
                // if stack is empty,
                // push the element
                if (st.size() == 0)
                {
                    st.push(fnt);
                }

                // if top element is less than
                // element which need to be
                // pushed, then return false.
                else if (st.size() != 0 &&
```



```
        st.peek() < fnt)
    {
        return false;
    }

    // else push into the stack.
    else
        st.push(fnt);
}

// while expected element are
// coming from stack, pop them out.
while (st.size() != 0 &&
        st.peek() == expected)
{
    st.pop();
    expected++;
}

// if the final expected element
// value is equal to initial Queue
// size and the stack is empty.
if (expected - 1 == n &&
    st.size() == 0)
    return true;

return false;
}

// Driver Code
public static void main(String args[])
{
    q.add(5);
    q.add(1);
    q.add(2);
    q.add(3);
    q.add(4);

    int n = q.size();

    if (checkSorted(n))
        System.out.print("Yes");
    else
        System.out.print("No");
}
}
```

```
// This code is contributed by  
// Manish Shaw(manishshaw1)
```

C#

```
// C# Program to check if a queue  
// of first n natural number can  
// be sorted using a stack  
using System;  
using System.Linq;  
using System.Collections.Generic;  
  
class GFG  
{  
    // Function to check if given  
    // queue element can be sorted  
    // into another queue using a stack.  
    static bool checkSorted(int n,  
                             ref Queue<int> q)  
    {  
        Stack<int> st = new Stack<int>();  
        int expected = 1;  
        int fnt;  
  
        // while given Queue  
        // is not empty.  
        while (q.Count != 0)  
        {  
            fnt = q.Peek();  
            q.Dequeue();  
  
            // if front element is  
            // the expected element  
            if (fnt == expected)  
                expected++;  
  
            else  
            {  
                // if stack is empty,  
                // push the element  
                if (st.Count != 0)  
                {  
                    st.Push(fnt);  
                }  
  
                // if top element is less than  
                // element which need to be  
                // pushed, then return false.
```

```
        else if (st.Count != 0 &&
                st.Peek() < fnt)
        {
            return false;
        }

        // else push into the stack.
        else
            st.Push(fnt);
    }

    // while expected element are
    // coming from stack, pop them out.
    while (st.Count != 0 &&
            st.Peek() == expected)
    {
        st.Pop();
        expected++;
    }
}

// if the final expected element
// value is equal to initial Queue
// size and the stack is empty.
if (expected - 1 == n &&
    st.Count == 0)
    return true;

return false;
}

// Driver Code
static void Main()
{
    Queue<int> q = new Queue<int>();
    q.Enqueue(5);
    q.Enqueue(1);
    q.Enqueue(2);
    q.Enqueue(3);
    q.Enqueue(4);

    int n = q.Count;

    if (checkSorted(n, ref q))
        Console.WriteLine("Yes");
    else
        Console.WriteLine("No");
}
}
```

```
// This code is contributed by  
// Manish Shaw(manishshaw1)
```

Output :

Yes

Video Contributed by [Parul Shandilya](#)

Improved By : [manishshaw1](#), [ParulShandilya](#), [Abdul Mohsin](#)

Source

<https://www.geeksforgeeks.org/check-queue-can-sorted-another-queue-using-stack/>

Chapter 11

Check if an array is stack sortable

Check if an array is stack sortable - GeeksforGeeks

Given an array of N distinct elements where elements are between 1 and N both inclusive, check if it is stack-sortable or not. An array A[] is said to be stack sortable if it can be stored in another array B[], using a temporary stack S.

The operations that are allowed on array are:

1. Remove the starting element of array A[] and push it into the stack.
2. Remove the top element of the stack S and append it to the end of array B.

If all the element of A[] can be moved to B[] by performing these operations such that array B is sorted in ascending order, then array A[] is stack sortable.

Examples:

Input : A[] = { 3, 2, 1 }

Output : YES

Explanation :

Step 1: Remove the starting element of array A[]
and push it in the stack S. (Operation 1)
That makes A[] = { 2, 1 } ; Stack S = { 3 }

Step 2: Operation 1
That makes A[] = { 1 } Stack S = { 3, 2 }

Step 3: Operation 1
That makes A[] = {} Stack S = { 3, 2, 1 }

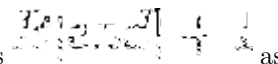
Step 4: Operation 2
That makes Stack S = { 3, 2 } B[] = { 1 }

Step 5: Operation 2

That makes Stack $S = \{ 3 \}$ $B[] = \{ 1, 2 \}$
Step 6: Operation 2
That makes Stack $S = \{ \}$ $B[] = \{ 1, 2, 3 \}$

Input : $A[] = \{ 2, 3, 1 \}$
Output : NO

Given, array $A[]$ is a permutation of $[1, \dots, N]$, so let us suppose the initially $B[] = \{0\}$. Now we can observe that:

1. We can only push an element in the stack S if the stack is empty or the current element is less than the top of the stack.
2. We can only pop from the stack only if the top of the stack is  as the array $B[]$ will contain $\{1, 2, 3, 4, \dots, n\}$.

Source

<https://www.geeksforgeeks.org/check-array-stack-sortable/>

Chapter 12

Check if concatenation of two strings is balanced or not

Check if concatenation of two strings is balanced or not - GeeksforGeeks

Given two bracket sequences S1 and S2 consisting of '(' and ')'. The task is to check if the string obtained by concatenating both the sequences is balanced or not. Concatenation can be done by s1+s2 or s2+s1.

Examples:

Input: s1 = ")()()())", s2 = "(()()("

Output: Balanced

s2 + s1 = "(()()()()())", which is a balanced parenthesis sequence.

Input: s1 = "(()))(", s2 = "())()"

Output: Not balanced

s1 + s2 = "(()))()())" -> Not balanced

s2 + s1 = "())()()())" -> Not balanced

A **naive** solution is to first concatenate both sequences and then check if the resultant sequence is balanced or not using a stack. First, check if s1 + s2 is balanced or not. If not, then check if s2 + s1 is balanced or not. To check if a given sequence of brackets is balanced or not using a stack, the following algorithm can be used.

1. Declare a character stack S.
2. Now traverse the expression string exp.
 - If the current character is a starting bracket ('(' or '{' or '[') then push it to stack.
 - If the current character is a closing bracket (')' or '}' or ']') then pop from the 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”.

Below is the implementation of above approach:

C++

```
// CPP program to check if sequence obtained
// by concatenating two bracket sequences
// is balanced or not.
#include <bits/stdc++.h>
using namespace std;

// Check if given string is balanced bracket
// sequence or not.
bool isBalanced(string s)
{
    stack<char> st;

    int n = s.length();

    for (int i = 0; i < n; i++) {

        // If current bracket is an opening
        // bracket push it to stack.
        if (s[i] == '(')
            st.push(s[i]);

        // If current bracket is a closing
        // bracket then pop from stack if
        // it is not empty. If stack is empty
        // then sequence is not balanced.
        else {
            if (st.empty()) {
                return false;
            }
            else
                st.pop();
        }
    }

    // If stack is not empty, then sequence
    // is not balanced.
    if (!st.empty())
        return false;
}
```



```
        return true;
    }

    // Function to check if string obtained by
    // concatenating two bracket sequences is
    // balanced or not.
    bool isBalancedSeq(string s1, string s2)
    {

        // Check if s1 + s2 is balanced or not.
        if (isBalanced(s1 + s2))
            return true;

        // Check if s2 + s1 is balanced or not.
        return isBalanced(s2 + s1);
    }

    // Driver code.
    int main()
    {
        string s1 = ")()((())))";
        string s2 = "(()(()(";

        if (isBalancedSeq(s1, s2))
            cout << "Balanced";
        else
            cout << "Not Balanced";

        return 0;
    }
```

Output:

Balanced

Time complexity: $O(n)$

Auxiliary Space: $O(n)$

An **efficient** solution is to check if given sequences can result in balanced parenthesis sequence without using a stack, i.e., in constant extra space.

Let the concatenated sequence be s . There are two possibilities: either $s = s1 + s2$ is balanced or $s = s2 + s1$ is balanced. Check for both possibilities whether s is balanced or not.

- If s is balanced, then the number of opening brackets in s should always be greater than or equal to the number of closing brackets in S at any instant of traversing it. This is because if at any instant number of closing brackets in s is greater than the

number of opening brackets, then the last closing bracket will not have a matching opening bracket (that is why the count is more) in s.

- If the sequence is balanced then at the end of traversal, the number of opening brackets in s is equal to the number of closing brackets in s.

Below is the implementation of above approach:

C++

```
// C++ program to check if sequence obtained
// by concatenating two bracket sequences
// is balanced or not.
#include <bits/stdc++.h>
using namespace std;

// Check if given string is balanced bracket
// sequence or not.
bool isBalanced(string s)
{
    // To store result of comparison of
    // count of opening brackets and
    // closing brackets.
    int cnt = 0;

    int n = s.length();

    for (int i = 0; i < n; i++) {

        // If current bracket is an
        // opening bracket, then
        // increment count.
        if (s[i] == '(')
            cnt++;

        // If current bracket is a
        // closing bracket, then
        // decrement count and check
        // if count is negative.
        else {
            cnt--;
            if (cnt < 0)
                return false;
        }
    }

    // If count is positive then
```

```
// some opening brackets are
// not balanced.
if (cnt > 0)
    return false;

return true;
}

// Function to check if string obtained by
// concatenating two bracket sequences is
// balanced or not.
bool isBalancedSeq(string s1, string s2)
{

    // Check if s1 + s2 is balanced or not.
    if (isBalanced(s1 + s2))
        return true;

    // Check if s2 + s1 is balanced or not.
    return isBalanced(s2 + s1);
}

// Driver code.
int main()
{
    string s1 = ")()((())))";
    string s2 = "(()(((";

    if (isBalancedSeq(s1, s2))
        cout << "Balanced";
    else
        cout << "Not Balanced";

    return 0;
}
```

Java

```
// Java program to check if
// sequence obtained by
// concatenating two bracket
// sequences is balanced or not.
import java.io.*;

class GFG
{

    // Check if given string
```

```
// is balanced bracket
// sequence or not.
static boolean isBalanced(String s)
{

    // To store result of comparison
    // of count of opening brackets
    // and closing brackets.
    int cnt = 0;
    int n = s.length();
    for (int i = 0; i < n; i++)
    {

        // If current bracket is
        // an opening bracket,
        // then increment count.
        if (s.charAt(i) == '(')
        {
            cnt = cnt + 1;
        }

        // If current bracket is a
        // closing bracket, then
        // decrement count and check
        // if count is negative.
        else
        {
            cnt = cnt - 1;
            if (cnt < 0)
                return false;
        }
    }

    // If count is positive then
    // some opening brackets are
    // not balanced.
    if (cnt > 0)
        return false;

    return true;
}

// Function to check if string
// obtained by concatenating
// two bracket sequences is
// balanced or not.
static boolean isBalancedSeq(String s1,
                             String s2)
```

```
{

// Check if s1 + s2 is
// balanced or not.
if (isBalanced(s1 + s2))
    return true;

// Check if s2 + s1 is
// balanced or not.
return isBalanced(s2 + s1);
}

// Driver code
public static void main(String [] args)
{
    String s1 = ")()((()))";
    String s2 = "(()(()(";

    if (isBalancedSeq(s1, s2))
    {
        System.out.println("Balanced");
    }
    else
    {
        System.out.println("Not Balanced");
    }
}
}

// This code is contributed
// by Shivi_Aggarwal
```

Python3

```
# Python3 program to check
# if sequence obtained by
# concatenating two bracket
# sequences is balanced or not.

# Check if given string
# is balanced bracket
# sequence or not.
def isBalanced(s):

    # To store result of
    # comparison of count
    # of opening brackets
    # and closing brackets.
```

```
cnt = 0
n = len(s)

for i in range(0, n):
    if (s[i] == '('):
        cnt = cnt + 1
    else :
        cnt = cnt - 1
        if (cnt < 0):
            return False
if (cnt > 0):
    return False

return True

def isBalancedSeq(s1, s2):

    if (isBalanced(s1 + s2)):
        return True

    return isBalanced(s2 + s1)

# Driver code
a = ")()()())";
b = "(()(()(";

if (isBalancedSeq(a, b)):
    print("Balanced")
else:
    print("Not Balanced")

# This code is contributed
# by Shivi_Aggarwal
```

Output:

Balanced

Time complexity: $O(n)$

Auxiliary Space: $O(1)$

Improved By : [Shivi_Aggarwal](#)

Source

<https://www.geeksforgeeks.org/check-if-concatenation-of-two-strings-is-balanced-or-not/>

Chapter 13

Check if two expressions with brackets are same

Check if two expressions with brackets are same - GeeksforGeeks

Given two expressions in the form of strings. The task is to compare them and check if they are similar. Expressions consist of lowercase alphabets, '+', '-', and '()'.

Examples:

```
Input   : exp1 = "-(a+b+c)"
          exp2 = "-a-b-c"
Output  : Yes
```

```
Input   : exp1 = "-(c+b+a)"
          exp2 = "-c-b-a"
Output  : Yes
```

```
Input   : exp1 = "a-b-(c-d)"
          exp2 = "a-b-c-d"
Output  : No
```

It may be assumed that there are at most 26 operands from 'a' to 'z' and every operand appears only once.

A simple idea behind is to keep a record of the *Global and Local Sign(+/-)* through the expression. The Global Sign here means the multiplicative sign at each operand. The resultant sign for an operand is local sign multiplied by the global sign at that operand.

For example, the expression $a+b-(c-d)$ is evaluated as $(+)+a(+)+b(-)+c(-)-d \Rightarrow a + b - c + d$. The global sign (represented inside bracket) is multiplied to the local sign for each operand.

In the given solution, stack is used to keep record of the global signs. A count vector records the counts of the operands(lowercase Latin letters here). Two expressions are evaluated in opposite manners and finally, it is checked if the all entries in the count vector are zeros.

```
// CPP program to check if two expressions
// evaluate to same.
#include <bits/stdc++.h>
using namespace std;

const int MAX_CHAR = 26;

// Return local sign of the operand. For example,
// in the expr a-b-(c), local signs of the operands
// are +a, -b, +c
bool adjSign(string s, int i)
{
    if (i == 0)
        return true;
    if (s[i - 1] == '-')
        return false;
    return true;
};

// Evaluate expressions into the count vector of
// the 26 alphabets.If add is true, then add count
// to the count vector of the alphabets, else remove
// count from the count vector.
void eval(string s, vector<int>& v, bool add)
{
    // stack stores the global sign
    // for operands.
    stack<bool> stk;
    stk.push(true);

    // + means true
    // global sign is positive initially

    int i = 0;
    while (s[i] != '\0') {
        if (s[i] == '+' || s[i] == '-') {
            i++;
            continue;
        }
        if (s[i] == '(') {

            // global sign for the bracket is
            // pushed to the stack
            if (adjSign(s, i))
```



```
        stk.push(stk.top());
    else
        stk.push(!stk.top());
}

// global sign is popped out which
// was pushed in for the last bracket
else if (s[i] == ')')
    stk.pop();

else {

    // global sign is positive (we use different
    // values in two calls of functions so that
    // we finally check if all vector elements
    // are 0.
    if (stk.top())
        v[s[i] - 'a'] += (adjSign(s, i) ? add ? 1 : -1 :
                           add ? -1 : 1);

    // global sign is negative here
    else
        v[s[i] - 'a'] += (adjSign(s, i) ? add ? -1 : 1 :
                           add ? 1 : -1);
}
i++;
}
};

// Returns true if expr1 and expr2 represent
// same expressions
bool areSame(string expr1, string expr2)
{
    // Create a vector for all operands and
    // initialize the vector as 0.
    vector<int> v(MAX_CHAR, 0);

    // Put signs of all operands in expr1
    eval(expr1, v, true);

    // Subtract signs of operands in expr2
    eval(expr2, v, false);

    // If expressions are same, vector must
    // be 0.
    for (int i = 0; i < MAX_CHAR; i++)
        if (v[i] != 0)
            return false;
}
```

```
        return true;
    }

    // Driver code
    int main()
    {
        string expr1 = "-(a+b+c)", expr2 = "-a-b-c";
        if (areSame(expr1, expr2))
            cout << "Yes\n";
        else
            cout << "No\n";
        return 0;
    }
```

Output:

YES

Source

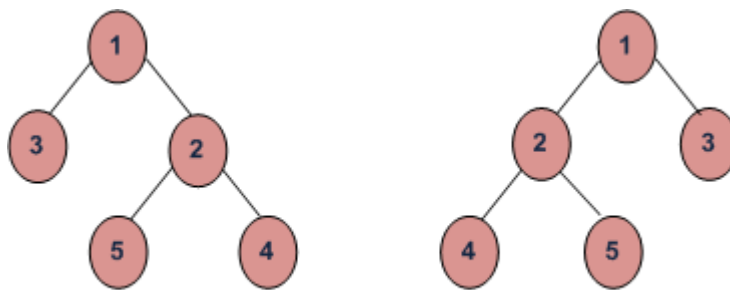
<https://www.geeksforgeeks.org/check-two-expressions-brackets/>

Chapter 14

Check if two trees are Mirror

Check if two trees are Mirror - GeeksforGeeks

Given two Binary Trees, write a function that returns true if two trees are mirror of each other, else false. For example, the function should return true for following input trees.



Mirror Trees

This problem is different from the problem discussed [here](#).

For two trees 'a' and 'b' to be mirror images, the following three conditions must be true:

1. Their root node's key must be same
2. Left subtree of root of 'a' and right subtree root of 'b' are mirror.
3. Right subtree of 'a' and left subtree of 'b' are mirror.

Below is implementation of above idea.

C++

```
// C++ program to check if two trees are mirror
// of each other
#include<bits/stdc++.h>
using namespace std;
```

```
/* A binary tree node has data, pointer to
   left child and a pointer to right child */
struct Node
{
    int data;
    Node* left, *right;
};

/* Given two trees, return true if they are
   mirror of each other */
int areMirror(Node* a, Node* b)
{
    /* Base case : Both empty */
    if (a==NULL && b==NULL)
        return true;

    // If only one is empty
    if (a==NULL || b == NULL)
        return false;

    /* Both non-empty, compare them recursively
       Note that in recursive calls, we pass left
       of one tree and right of other tree */
    return a->data == b->data &&
           areMirror(a->left, b->right) &&
           areMirror(a->right, b->left);
}

/* Helper function that allocates a new node */
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return(node);
}

/* Driver program to test areMirror() */
int main()
{
    Node *a = newNode(1);
    Node *b = newNode(1);
    a->left = newNode(2);
    a->right = newNode(3);
    a->left->left = newNode(4);
    a->left->right = newNode(5);
```

```
b->left = newNode(3);
b->right = newNode(2);
b->right->left = newNode(5);
b->right->right = newNode(4);

areMirror(a, b)? cout << "Yes" : cout << "No";

return 0;
}
```

Java

```
// Java program to see if two trees
// are mirror of each other

// A binary tree node
class Node
{
    int data;
    Node left, right;

    public Node(int data)
    {
        this.data = data;
        left = right = null;
    }
}

class BinaryTree
{
    Node a, b;

    /* Given two trees, return true if they are
       mirror of each other */
    boolean areMirror(Node a, Node b)
    {
        /* Base case : Both empty */
        if (a == null && b == null)
            return true;

        // If only one is empty
        if (a == null || b == null)
            return false;

        /* Both non-empty, compare them recursively
           Note that in recursive calls, we pass left
           of one tree and right of other tree */
        return a.data == b.data
    }
}
```

```
        && areMirror(a.left, b.right)
        && areMirror(a.right, b.left);
    }

    // Driver code to test above methods
    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree();
        Node a = new Node(1);
        Node b = new Node(1);
        a.left = new Node(2);
        a.right = new Node(3);
        a.left.left = new Node(4);
        a.left.right = new Node(5);

        b.left = new Node(3);
        b.right = new Node(2);
        b.right.left = new Node(5);
        b.right.right = new Node(4);

        if (tree.areMirror(a, b) == true)
            System.out.println("Yes");
        else
            System.out.println("No");
    }
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Python3

```
# Python3 program to check if two
# trees are mirror of each other

# A binary tree node
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Given two trees, return true
# if they are mirror of each other
def areMirror(a, b):

    # Base case : Both empty
    if a is None and b is None:
```

```
        return True

    # If only one is empty
    if a is None or b is None:
        return False

    # Both non-empty, compare them
    # recursively. Note that in
    # recursive calls, we pass left
    # of one tree and right of other tree
    return (a.data == b.data and
            areMirror(a.left, b.right) and
            areMirror(a.right , b.left))

# Driver code
root1 = Node(1)
root2 = Node(1)

root1.left = Node(2)
root1.right = Node(3)
root1.left.left = Node(4)
root1.left.right = Node(5)

root2.left = Node(3)
root2.right = Node(2)
root2.right.left = Node(5)
root2.right.right = Node(4)

if areMirror(root1, root2):
    print ("Yes")
else:
    print ("No")

# This code is contributed by AshishR
```

Output :

Yes

Time Complexity : $O(n)$

Iterative method to check if two trees are mirror of each other

This article is contributed by **Ashish Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : [AshishR](#)

Source

<https://www.geeksforgeeks.org/check-if-two-trees-are-mirror/>

Chapter 15

Check mirror in n-ary tree

Check mirror in n-ary tree - GeeksforGeeks

Given two n-ary trees, the task is to check if they are mirror of each other or not. Print “Yes” if they are mirror of each other else “No”.

Examples:

```
Input : Node = 3, Edges = 2
Edge 1 of first N-ary: 1 2
Edge 2 of first N-ary: 1 3
Edge 1 of second N-ary: 1 2
Edge 2 of second N-ary: 1 3
Output : Yes
```

```
Input : Node = 3, Edges = 2
Edge 1 of first N-ary: 1 2
Edge 2 of first N-ary: 1 3
Edge 1 of second N-ary: 1 2
Edge 2 of second N-ary: 1 3
Output : No
```

The idea is to use Queue and Stack to check if given N-ary tree are mirror of each other or not.

Let first n-ary tree be t1 and second n-ary tree is t2. For each node in t1, make stack and push its connected node in it. Now, for each node in t2, make queue and push its connected node in it.

Now, for each corresponding node do following:

```
While stack and Queue is not empty.
a = top element of stack;
```

```

    b = front of stack;
    if (a != b)
        return false;
    pop element from stack and queue.

// C++ program to check if two n-ary trees are
// mirror.
#include <bits/stdc++.h>
using namespace std;

// First vector stores all nodes and adjacent of every
// node in a stack.
// Second vector stores all nodes and adjacent of every
// node in a queue.
bool mirrorUtil(vector<stack<int> >& tree1,
                vector<queue<int> >& tree2)
{
    // Traversing each node in tree.
    for (int i = 1; i < tree1.size(); ++i) {
        stack<int>& s = tree1[i];
        queue<int>& q = tree2[i];

        // While stack is not empty && Queue is not empty
        while (!s.empty() && !q.empty()) {

            // checking top element of stack and front
            // of queue.
            if (s.top() != q.front())
                return false;

            s.pop();
            q.pop();
        }

        // If queue or stack is not empty, return false.
        if (!s.empty() || !q.empty())
            return false;
    }

    return true;
}

// Returns true if given two trees are mirrors.
// A tree is represented as two arrays to store
// all tree edges.
void areMirrors(int m, int n, int u1[], int v1[],
                int u2[], int v2[])
{

```

```
vector<stack<int> > tree1(m + 1);
vector<queue<int> > tree2(m + 1);

// Pushing node in the stack of first tree.
for (int i = 0; i < n; i++)
    tree1[u1[i]].push(v1[i]);

// Pushing node in the queue of second tree.
for (int i = 0; i < n; i++)
    tree2[u2[i]].push(v2[i]);

mirrorUtil(tree1, tree2) ? (cout << "Yes" << endl) :
                           (cout << "No" << endl);
}

// Driver code
int main()
{
    int M = 3, N = 2;

    int u1[] = { 1, 1 };
    int v1[] = { 2, 3 };

    int u2[] = { 1, 1 };
    int v2[] = { 3, 2 };

    areMirrors(M, N, u1, v1, u2, v2);

    return 0;
}
```

Output:

Yes

Reference: <https://practice.geeksforgeeks.org/problems/check-mirror-in-n-ary-tree/0>

Source

<https://www.geeksforgeeks.org/check-mirror-n-ary-tree/>

Chapter 16

Computer Organization | Stack based CPU Organization

Computer Organization | Stack based CPU Organization - GeeksforGeeks

The computers which use Stack based CPU Organization are based on a data structure called **stack**. Stack is a list of data words. It uses **Last In First Out (LIFO)** access method which is the most popular access method in most of the CPU. A register is used to store the address of the top most element of the stack which is known as **Stack pointer (SP)**.

The main two operations that are performed on the operators of the stack are **Push** and **Pop**. These two operations are performed from one end only.

1. **Push** –

This operation results in inserting one operand at the top of the stack and it decreases the stack pointer register. The format of the PUSH instruction is:

PUSH

It inserts the data word at specified address to the top of the stack. It can be implemented as:

```
//decrement SP by 1
SP <-- SP - 1

//store the content of specified memory address
//into SP; i.e, at top of stack
SP <-- (memory address)
```

2. **Pop** –

This operation results in deleting one operand from the top of the stack and it

increase the stack pointer register. The format of the POP instruction is:

POP

It deletes the data word at the top of the stack to the specified address. It can be implemented as:

```
//transfer the content of SP (i.e, at top most data)
//into specified memory location
(memory address) <-- SP

//increment SP by 1
SP <-- SP + 1
```

Operation type instruction do not need address field in this CPU organization. This is because the operation is performed on the two operands that are on the top of the stack. For example:

SUB

This instruction contains the opcode only with no address field. It pops the two top data from the stack, subtracting the data, and pushing the result into the stack at the top.

PDP-11, Intel's 8085 and HP 3000 are some of the examples of the stack organized computers.

The advantages of Stack based CPU organization –

- Efficient computation of complex arithmetic expressions.
- Execution of instructions is fast because operand data are stored in consecutive memory locations.
- Length of instruction is short as they do not have address field.

The disadvantages of Stack based CPU organization –

- The size of the program increases.

Source

<https://www.geeksforgeeks.org/computer-organization-stack-based-cpu-organization/>

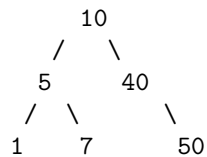
Chapter 17

Construct BST from given preorder traversal | Set 2

Construct BST from given preorder traversal | Set 2 - GeeksforGeeks

Given preorder traversal of a binary search tree, construct the BST.

For example, if the given traversal is {10, 5, 1, 7, 40, 50}, then the output should be root of following tree.



We have discussed $O(n^2)$ and $O(n)$ recursive solutions in the [previous post](#). Following is a stack based iterative solution that works in $O(n)$ time.

1. Create an empty stack.
2. Make the first value as root. Push it to the stack.
3. Keep on popping while the stack is not empty and the next value is greater than stack's top value. Make this value as the right child of the last popped node. Push the new node to the stack.
4. If the next value is less than the stack's top value, make this value as the left child of the stack's top node. Push the new node to the stack.
5. Repeat steps 2 and 3 until there are items remaining in `pre[]`.

C

```
/* A O(n) iterative program for construction of BST from preorder traversal */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
typedef struct Node
{
    int data;
    struct Node *left, *right;
} Node;

// A Stack has array of Nodes, capacity, and top
typedef struct Stack
{
    int top;
    int capacity;
    Node* *array;
} Stack;

// A utility function to create a new tree node
Node* newNode( int data )
{
    Node* temp = (Node *)malloc( sizeof( Node ) );
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to create a stack of given capacity
Stack* createStack( int capacity )
{
    Stack* stack = (Stack *)malloc( sizeof( Stack ) );
    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (Node **)malloc( stack->capacity * sizeof( Node* ) );
    return stack;
}

// A utility function to check if stack is full
int isFull( Stack* stack )
{
    return stack->top == stack->capacity - 1;
}

// A utility function to check if stack is empty
int isEmpty( Stack* stack )
```

```
{
    return stack->top == -1;
}

// A utility function to push an item to stack
void push( Stack* stack, Node* item )
{
    if( isFull( stack ) )
        return;
    stack->array[ ++stack->top ] = item;
}

// A utility function to remove an item from stack
Node* pop( Stack* stack )
{
    if( isEmpty( stack ) )
        return NULL;
    return stack->array[ stack->top-- ];
}

// A utility function to get top node of stack
Node* peek( Stack* stack )
{
    return stack->array[ stack->top ];
}

// The main function that constructs BST from pre[]
Node* constructTree ( int pre[], int size )
{
    // Create a stack of capacity equal to size
    Stack* stack = createStack( size );

    // The first element of pre[] is always root
    Node* root = newNode( pre[0] );

    // Push root
    push( stack, root );

    int i;
    Node* temp;

    // Iterate through rest of the size-1 items of given preorder array
    for ( i = 1; i < size; ++i )
    {
        temp = NULL;

        /* Keep on popping while the next value is greater than
           stack's top value. */
    }
}
```



```
while ( !isEmpty( stack ) && pre[i] > peek( stack )->data )
    temp = pop( stack );

// Make this greater value as the right child and push it to the stack
if ( temp != NULL)
{
    temp->right = newNode( pre[i] );
    push( stack, temp->right );
}

// If the next value is less than the stack's top value, make this value
// as the left child of the stack's top node. Push the new node to stack
else
{
    peek( stack )->left = newNode( pre[i] );
    push( stack, peek( stack )->left );
}
}

return root;
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder (Node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

// Driver program to test above functions
int main ()
{
    int pre[] = {10, 5, 1, 7, 40, 50};
    int size = sizeof( pre ) / sizeof( pre[0] );

    Node *root = constructTree(pre, size);

    printf("Inorder traversal of the constructed tree: \n");
    printInorder(root);

    return 0;
}
```

Java

```
// Java program to construct BST from given preorder traversal

import java.util.*;

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class BinaryTree {

    // The main function that constructs BST from pre[]
    Node constructTree(int pre[], int size) {

        // The first element of pre[] is always root
        Node root = new Node(pre[0]);

        Stack<Node> s = new Stack<Node>();

        // Push root
        s.push(root);

        // Iterate through rest of the size-1 items of given preorder array
        for (int i = 1; i < size; ++i) {
            Node temp = null;

            /* Keep on popping while the next value is greater than
            stack's top value. */
            while (!s.isEmpty() && pre[i] > s.peek().data) {
                temp = s.pop();
            }

            // Make this greater value as the right child and push it to the stack
            if (temp != null) {
                temp.right = new Node(pre[i]);
                s.push(temp.right);
            }

            // If the next value is less than the stack's top value, make this value
            // as the left child of the stack's top node. Push the new node to stack
            else {
```

```
        temp = s.peek();
        temp.left = new Node(pre[i]);
        s.push(temp.left);
    }
}

return root;
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder(Node node) {
    if (node == null) {
        return;
    }
    printInorder(node.left);
    System.out.print(node.data + " ");
    printInorder(node.right);
}

// Driver program to test above functions
public static void main(String[] args) {
    BinaryTree tree = new BinaryTree();
    int pre[] = new int[]{10, 5, 1, 7, 40, 50};
    int size = pre.length;
    Node root = tree.constructTree(pre, size);
    System.out.println("Inorder traversal of the constructed tree is ");
    tree.printInorder(root);
}

// This code has been contributed by Mayank Jaiswal
```

Output:

1 5 7 10 40 50

Time Complexity: $O(n)$. The complexity looks more from first look. If we take a closer look, we can observe that every item is pushed and popped only once. So at most $2n$ push/pop operations are performed in the main loops of `constructTree()`. Therefore, time complexity is $O(n)$.

Source

<https://www.geeksforgeeks.org/construct-bst-from-given-preorder-traversal-set-2/>

Chapter 18

Construct Binary Tree from String with bracket representation

Construct Binary Tree from String with bracket representation - GeeksforGeeks

Construct a binary tree from a string consisting of parenthesis and integers. The whole input represents a binary tree. It contains an integer followed by zero, one or two pairs of parenthesis. The integer represents the root's value and a pair of parenthesis contains a child binary tree with the same structure. Always start to construct the left child node of the parent first if it exists.

Examples:

Input : "1(2)(3)"

Output : 1 2 3

Explanation :

```
      1
     / \
    2   3
```

Explanation: first pair of parenthesis contains left subtree and second one contains the right subtree. Preorder of above tree is "1 2 3".

Input : "4(2(3)(1))(6(5))"

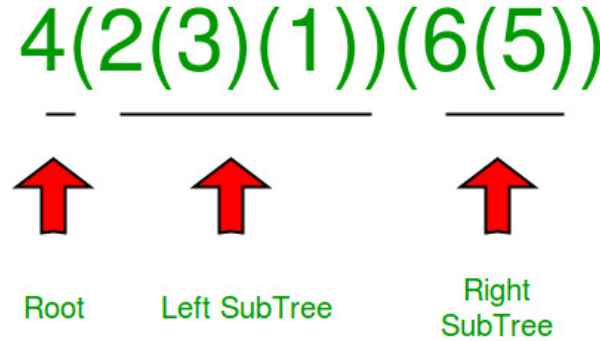
Output : 4 2 3 1 6 5

Explanation :

```
      4
     / \
    2   6
   /\  /
  3 1 5
```

3 1 5

We know first character in string is root. Substring inside the first adjacent pair of parenthesis is for left subtree and substring inside second pair of parenthesis is for right subtree as in the below diagram.



We need to find the substring corresponding to left subtree and substring corresponding to right subtree and then recursively call on both of the substrings.

For this first find the index of starting index and end index of each substring.

To find the index of closing parenthesis of left subtree substring, use a stack. Let's the found index is stored in index variable.

```

/* C++ program to construct a binary tree from
   the given string */
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left
   child and a pointer to right child */
struct Node {
    int data;
    Node *left, *right;
};

/* Helper function that allocates a new node */
Node* newNode(int data)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

/* This function is here just to test */
void preOrder(Node* node)
{
    if (node == NULL)

```

```

        return;
    printf("%d ", node->data);
    preOrder(node->left);
    preOrder(node->right);
}

// functin to return the index of close parenthesis
int findIndex(string str, int si, int ei)
{
    if (si > ei)
        return -1;

    // Inbuilt stack
    stack<char> s;

    for (int i = si; i <= ei; i++) {

        // if open parenthesis, push it
        if (str[i] == '(')
            s.push(str[i]);

        // if close parenthesis
        else if (str[i] == ')') {
            if (s.top() == '(') {
                s.pop();

                // if stack is empty, this is
                // the required index
                if (s.empty())
                    return i;
            }
        }
    }

    // if not found return -1
    return -1;
}

// function to construct tree from string
Node* treeFromString(string str, int si, int ei)
{
    // Base case
    if (si > ei)
        return NULL;

    // new root
    Node* root = newNode(str[si] - '0');
    int index = -1;

```

```
// if next char is '(' find the index of
// its complement ')'
if (si + 1 <= ei && str[si + 1] == '(')
    index = findIndex(str, si + 1, ei);

// if index found
if (index != -1) {

    // call for left subtree
    root->left = treeFromString(str, si + 2, index - 1);

    // call for right subtree
    root->right = treeFromString(str, index + 2, ei - 1);
}
return root;
}

// Driver Code
int main()
{
    string str = "4(2(3)(1))(6(5))";
    Node* root = treeFromString(str, 0, str.length() - 1);
    preOrder(root);
}
```

Output:

4 2 3 1 6 5

Source

<https://www.geeksforgeeks.org/construct-binary-tree-string-bracket-representation/>

Chapter 19

Convert Infix To Prefix Notation

Convert Infix To Prefix Notation - GeeksforGeeks

While we use infix expressions in our day to day lives. Computers have trouble understanding this format because they need to keep in mind rules of operator precedence and also brackets. Prefix and Postfix expressions are easier for a computer to understand and evaluate.

Given two operands a and b and an operator O , the infix notation implies that O will be placed in between a and b i.e. $a O b$. When the operator is placed after both operands i.e. abO , it is called postfix notation. And when the operator is placed before the operands i.e. Oab , the expression is in prefix notation.

Given any infix expression we can obtain the equivalent prefix and postfix format.

Examples:

Input : $A * B + C / D$

Output : $+ * A B / C D$

Input : $(A - B/C) * (A/K-L)$

Output : $*-A/BC-/AKL$

To convert an infix to postfix expression refer to this article [Stack | Set 2 \(Infix to Postfix\)](#). 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 ')' becomes '('.
- Step 2: Obtain the postfix expression of the modified expression i.e. $CB*A+$.

- Step 3: Reverse the postfix expression. Hence in our example prefix is +A*BC.

Below is the C++ implementation of the algorithm.

```
// CPP program to convert infix to prefix
#include <bits/stdc++.h>
using namespace std;

bool isOperator(char c)
{
    return (!isalpha(c) && !isdigit(c));
}

int getPriority(char C)
{
    if (C == '-' || C == '+')
        return 1;
    else if (C == '*' || C == '/')
        return 2;
    else if (C == '^')
        return 3;
    return 0;
}

string infixToPostfix(string infix)
{
    infix = '(' + infix + ')';
    int l = infix.size();
    stack<char> char_stack;
    string output;

    for (int i = 0; i < l; i++) {

        // If the scanned character is an
        // operand, add it to output.
        if (isalpha(infix[i]) || isdigit(infix[i]))
            output += infix[i];

        // If the scanned character is an
        // '(', push it to the stack.
        else if (infix[i] == '(')
            char_stack.push('(');

        // If the scanned character is an
        // ')', pop and output from the stack
        // until an '(' is encountered.
        else if (infix[i] == ')') {
```

```
        while (char_stack.top() != '(') {
            output += char_stack.top();
            char_stack.pop();
        }

        // Remove '(' from the stack
        char_stack.pop();
    }

    // Operator found
    else {

        if (isOperator(char_stack.top())) {
            while (getPriority(infix[i])
                <= getPriority(char_stack.top())) {
                output += char_stack.top();
                char_stack.pop();
            }

            // Push current Operator on stack
            char_stack.push(infix[i]);
        }
    }
}
return output;
}

string infixToPrefix(string infix)
{
    /* Reverse String
     * Replace ( with ) and vice versa
     * Get Postfix
     * Reverse Postfix * */
    int l = infix.size();

    // Reverse infix
    reverse(infix.begin(), infix.end());

    // Replace ( with ) and vice versa
    for (int i = 0; i < l; i++) {

        if (infix[i] == '(') {
            infix[i] = ')';
            i++;
        }
        else if (infix[i] == ')') {
            infix[i] = '(';
            i++;
        }
    }
}
```

```
    }  
}  
  
string prefix = infixToPostfix(infix);  
  
// Reverse postfix  
reverse(prefix.begin(), prefix.end());  
  
return prefix;  
}  
  
// Driver code  
int main()  
{  
    string s = "(a-b/c)*(a/k-l)";  
    cout << infixToPrefix(s) << std::endl;  
    return 0;  
}
```

Output:

*-a/bc-/akl

Complexity:

Stack operations like push() and pop() are performed in constant time. Since we scan all the characters in the expression once the complexity is linear in time i.e $O(n)$.

Source

<https://www.geeksforgeeks.org/convert-infix-prefix-notation/>

Chapter 20

Count natural numbers whose all permutation are greater than that number

Count natural numbers whose all permutation are greater than that number - GeeksforGeeks

There are some natural number whose all permutation is greater than or equal to that number eg. 123, whose all the permutation (123, 231, 321) are greater than or equal to 123.

Given a natural number **n**, the task is to count all such number from 1 to n.

Examples:

```
Input : n = 15.  
Output : 14  
1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12,  
13, 14, 15 are the numbers whose all  
permutation is greater than the number  
itself. So, output 14.
```

```
Input : n = 100.  
Output : 54
```

A **simple solution** is to run a loop from 1 to n and for every number check if its digits are in non-decreasing order or not.

An **efficient solution** is based on below observations.

Observation 1: From 1 to 9, all number have this property. So, for $n \leq 9$, output n.

Observation 2: The number whose all permutation is greater than or equal to that number have all their digits in increasing order.

The idea is to push all the number from 1 to 9. Now, pop the top element, say **topel** and try to make number whose digits are in increasing order and the first digit is **topel**. To make such numbers, the second digit can be from **topel%10** to 9. If this number is less than **n**, increment the count and push the number in the stack, else ignore.

Below is C++ implementation of this approach:

```
// C++ program to count the number less than N,  
// whose all permutation is greater than or equal to the number.  
#include<bits/stdc++.h>  
using namespace std;  
  
// Return the count of the number having all  
// permutation greater than or equal to the number.  
int countNumber(int n)  
{  
    int result = 0;  
  
    // Pushing 1 to 9 because all number from 1  
    // to 9 have this property.  
    for (int i = 1; i <= 9; i++)  
    {  
        stack<int> s;  
        if (i <= n)  
        {  
            s.push(i);  
            result++;  
        }  
  
        // take a number from stack and add  
        // a digit smaller than last digit  
        // of it.  
        while (!s.empty())  
        {  
            int tp = s.top();  
            s.pop();  
            for (int j = tp%10; j <= 9; j++)  
            {  
                int x = tp*10 + j;  
                if (x <= n)  
                {  
                    s.push(x);  
                    result++;  
                }  
            }  
        }  
    }  
    return result;  
}
```

```
}

// Driven Program
int main()
{
    int n = 15;
    cout << countNumber(n) << endl;
    return 0;
}
```

Output:

14

Time Complexity : $O(x)$ where x is number of elements printed in output.

Source

<https://www.geeksforgeeks.org/count-natural-numbers-whose-permutation-greater-number/>

Chapter 21

Count subarrays where second highest lie before highest

Count subarrays where second highest lie before highest - GeeksforGeeks

Given an array of N distinct element of at least size 2. A pair (a, b) in an array is defined as 'a' is the index of second smallest element and 'b' is the index of highest element in the array. The task is to count all the distinct pair where $a < b$ in all the subarrays.

Examples :

Input : `arr[] = { 1, 3, 2, 4 }`
Output : 3

Explanation :

The subarray { 1 }, { 3 }, { 2 }, { 4 } does not contain any such pair

The subarray { 1, 3 }, { 2, 4 } contain (1, 2) as pair

The subarray { 3, 2 } does not contain any such pair

The subarray { 3, 2, 4 } contain (1, 3) as a pair

The subarray { 1, 3, 2 } does not contain any such pair

The subarray { 1, 3, 2, 4 } contain (2, 4) as a pair

So, there are 3 distinct pairs, which are (1, 2), (1, 3) and (2, 4).

Method 1: (Brute Force) – Simple approach can be,

1. Find all the subarrays.
2. For each subarray, find second largest and largest element.
3. Check if second largest element lie before largest element.
4. If so, check if such index pair is already counted or not. If not store the pair and increment the count by 1, else ignore.

Time Complexity: $O(n^2)$.

Method 2: (Using stack) –

For given array A, suppose for an element at index *curr* (A[curr]), first element greater than

it and after it is $A[\text{next}]$ and first element greater than it and before it $A[\text{prev}]$. Observe that for all subarray starting from any index in range $[\text{prev} + 1, \text{curr}]$ and ending at index next, $A[\text{curr}]$ is the second largest and $A[\text{next}]$ is the largest, which generate $(\text{curr} - \text{prev} + 1)$ pairs in total with difference of $(\text{next} - \text{curr} + 1)$ in maximum and second maximum.

If we get next and prev greater element in an array, and keep track of maximum number of pairs possible for any difference (of largest and second largest). We will need to add all these numbers.

Now only job left is to get greater element (after and before) any element. For this, refer [Next Greater Element](#).

Traverse from the starting element in an array, keeping track of all numbers in the stack in decreasing order. After arriving at any number, pop all elements from stack which are less than current element to get location of number bigger than it and push current element on it. This generates required value for all numbers in the array.

```
// C++ program to count number of distinct instance
// where second highest number lie
// before highest number in all subarrays.
#include <bits/stdc++.h>
#define MAXN 100005
using namespace std;

// Finding the next greater element of the array.
void makeNext(int arr[], int n, int nextBig[])
{
    stack<pair<int, int> > s;

    for (int i = n - 1; i >= 0; i--) {

        nextBig[i] = i;
        while (!s.empty() && s.top().first < arr[i])
            s.pop();

        if (!s.empty())
            nextBig[i] = s.top().second;

        s.push(pair<int, int>(arr[i], i));
    }
}

// Finding the previous greater element of the array.
void makePrev(int arr[], int n, int prevBig[])
{
    stack<pair<int, int> > s;
    for (int i = 0; i < n; i++) {

        prevBig[i] = -1;
```



```

        while (!s.empty() && s.top().first < arr[i])
            s.pop();

        if (!s.empty())
            prevBig[i] = s.top().second;

        s.push(pair<int, int>(arr[i], i));
    }
}

// Wrapper Function
int wrapper(int arr[], int n)
{
    int nextBig[MAXN];
    int prevBig[MAXN];
    int maxi[MAXN];
    int ans = 0;

    // Finding previous largest element
    makePrev(arr, n, prevBig);

    // Finding next largest element
    makeNext(arr, n, nextBig);

    for (int i = 0; i < n; i++)
        if (nextBig[i] != i)
            maxi[nextBig[i] - i] = max(maxi[nextBig[i] - i],
                                       i - prevBig[i]);

    for (int i = 0; i < n; i++)
        ans += maxi[i];

    return ans;
}

// Driven Program
int main()
{
    int arr[] = { 1, 3, 2, 4 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << wrapper(arr, n) << endl;
    return 0;
}

```

Output :

Time Complexity: $O(n)$

Source

<https://www.geeksforgeeks.org/count-subarrays-second-highest-lie-highest/>

Chapter 22

Create a customized data structure which evaluates functions in $O(1)$

Create a customized data structure which evaluates functions in $O(1)$ - GeeksforGeeks

Create a customized data structure such that it has functions :-

GetLastElement();

RemoveLastElement();

AddElement();

GetMin();

All the functions should be of $O(1)$

Question Source : amazon interview questions

Approach :

- 1) create a custom stack of type structure with two elements, (element, min_till_now)
- 2) implement the functions on this custom data type

```
// program to demonstrate customized data structure
// which supports functions in  $O(1)$ 
#include <iostream>
#include <vector>
using namespace std;
const int MAXX = 1000;

// class stack
class stack {
    int minn;
    int size;
```

```
public:
    stack()
    {
        minn = 99999;
        size = -1;
    }
    vector<pair<int, int> > arr;
    int GetLastElement();
    int RemoveLastElement();
    int AddElement(int element);
    int GetMin();
};

// utility function for adding a new element
int stack::AddElement(int element)
{
    if (size > MAXX) {
        cout << "stack overflow, max size reached!\n";
        return 0;
    }
    if (element < minn)
        minn = element;
    arr.push_back(make_pair(element, minn));
    size++;
    return 1;
}

// utility function for returning last element of stack
int stack::GetLastElement()
{
    if (size == -1) {
        cout << "No elements in stack\n";
        return 0;
    }
    return arr[size].first;
}

// utility function for removing last element successfully;
int stack::RemoveLastElement()
{
    if (size == -1) {
        cout << "stack empty!!!\n";
        return 0;
    }

    // updating minimum element
    if (size > 0 && arr[size - 1].second > arr[size].second) {
        minn = arr[size - 1].second;
    }
}
```

```
    }
    arr.pop_back();
    size -= 1;
    return 1;
}

// utility function for returning min element till now;
int stack::GetMin()
{
    if (size == -1) {
        cout << "stack empty!!\n";
        return 0;
    }
    return arr[size].second;
}

// Driver code
int main()
{
    stack s;
    int success = s.AddElement(5);
    if (success == 1)
        cout << "5 inserted successfully\n";

    success = s.AddElement(7);
    if (success == 1)
        cout << "7 inserted successfully\n";

    success = s.AddElement(3);
    if (success == 1)
        cout << "3 inserted successfully\n";
    int min1 = s.GetMin();
    cout << "min element  :: " << min1 << endl;

    success = s.RemoveLastElement();
    if (success == 1)
        cout << "removed successfully\n";

    success = s.AddElement(2);
    if (success == 1)
        cout << "2 inserted successfully\n";

    success = s.AddElement(9);
    if (success == 1)
        cout << "9 inserted successfully\n";
    int last = s.GetLastElement();
    cout << "Last element :: " << last << endl;
```

```
    success = s.AddElement(0);
    if (success == 1)
        cout << "0 inserted successfully\n";
    min1 = s.GetMin();
    cout << "min element  :: " << min1 << endl;

    success = s.RemoveLastElement();
    if (success == 1)
        cout << "removed successfully\n";

    success = s.AddElement(11);
    if (success == 1)
        cout << "11 inserted successfully\n";
    min1 = s.GetMin();
    cout << "min element  :: " << min1 << endl;

    return 0;
}
```

Output:

```
5 inserted successfully
7 inserted successfully
3 inserted successfully
min element  :: 3
removed successfully
2 inserted successfully
9 inserted successfully
Last element :: 9
0 inserted successfully
min element  :: 0
removed successfully
11 inserted successfully
min element  :: 2
```

Time complexity : Each function runs in $O(1)$

Source

<https://www.geeksforgeeks.org/create-customized-data-structure-evaluates-functions-o1/>

Chapter 23

Decode a string recursively encoded as count followed by substring

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

An encoded string (s) is given, the task is to decode it. The pattern in which the strings are encoded is as follows.

```
<count>[sub_str] ==> The substring 'sub_str'  
                        appears count times.
```

Examples:

```
Input : str[] = "1[b]"  
Output : b
```

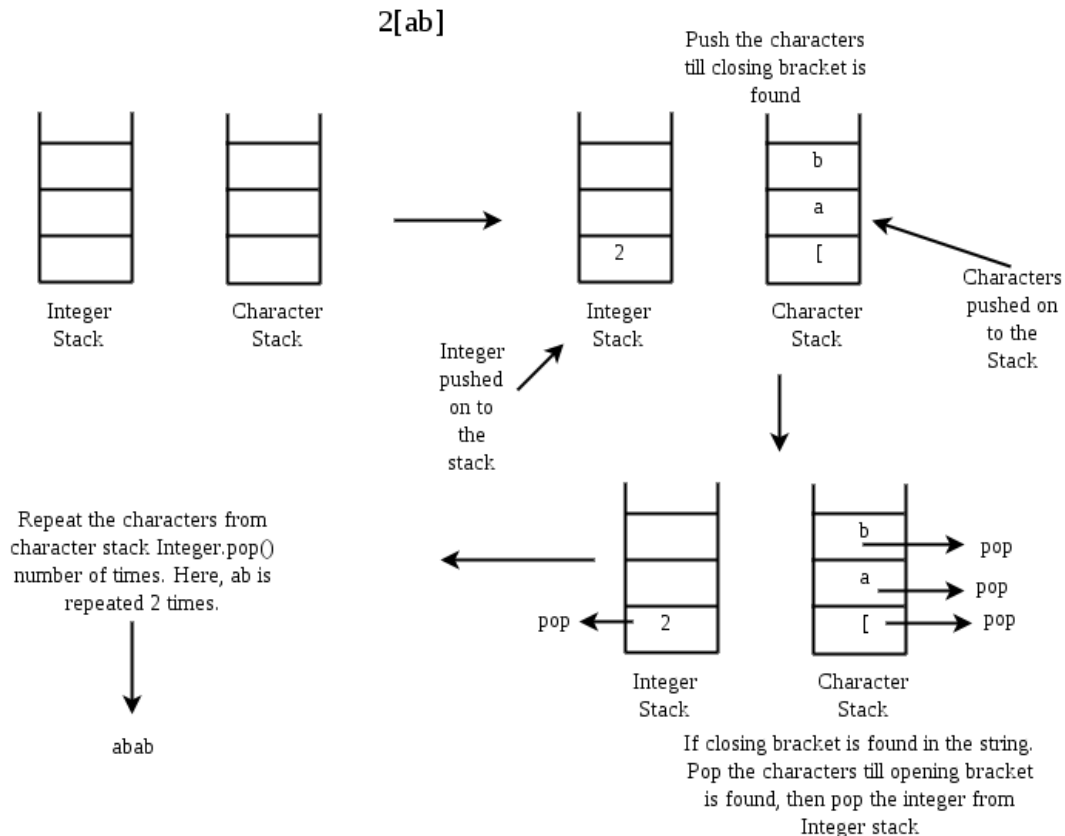
```
Input : str[] = "2[ab]"  
Output : abab
```

```
Input : str[] = "2[a2[b]]"  
Output : abbabb
```

```
Input : str[] = "3[b2[ca]]"  
Output : bcacabacabcaca
```

The idea is to use two stacks, one for integers and another for characters. Now, traverse the string,

1. Whenever we encounter any number, push it into the integer stack and in case of any alphabet (a to z) or open bracket ('['), push it onto the character stack.
2. Whenever any close bracket (']') is encountered pop the character from the character stack until open bracket ('[') is not found in the character stack. Also, pop the top element from the integer stack, say n. Now make a string repeating the popped character n number of times. Now, push all character of the string in the stack.



Below is implementation of this approach:

C++

```
// C++ program to decode a string recursively
// encoded as count followed substring
#include<bits/stdc++.h>
using namespace std;

// Returns decoded string for 'str'
string decode(string str)
{
    stack<int> integerstack;
```



```
stack<char> stringstack;
string temp = "", result = "";

// Traversing the string
for (int i = 0; i < str.length(); i++)
{
    int count = 0;

    // If number, convert it into number
    // and push it into integerstack.
    if (str[i] >= '0' && str[i] <='9')
    {
        while (str[i] >= '0' && str[i] <= '9')
        {
            count = count * 10 + str[i] - '0';
            i++;
        }

        i--;
        integerstack.push(count);
    }

    // If closing bracket ']', pop element until
    // '[' opening bracket is not found in the
    // character stack.
    else if (str[i] == ']')
    {
        temp = "";
        count = 0;

        if (! integerstack.empty())
        {
            count = integerstack.top();
            integerstack.pop();
        }

        while (! stringstack.empty() && stringstack.top() != '[' )
        {
            temp = stringstack.top() + temp;
            stringstack.pop();
        }

        if (! stringstack.empty() && stringstack.top() == '[')
            stringstack.pop();

        // Repeating the popped string 'temo' count
        // number of times.
        for (int j = 0; j < count; j++)
```

```
        result = result + temp;

        // Push it in the character stack.
        for (int j = 0; j < result.length(); j++)
            stringstack.push(result[j]);

        result = "";
    }

    // If '[' opening bracket, push it into character stack.
    else if (str[i] == '[')
    {
        if (str[i-1] >= '0' && str[i-1] <= '9')
            stringstack.push(str[i]);

        else
        {
            stringstack.push(str[i]);
            integerstack.push(1);
        }
    }

    else
        stringstack.push(str[i]);
}

// Pop all the elmenet, make a string and return.
while (! stringstack.empty())
{
    result = stringstack.top() + result;
    stringstack.pop();
}

return result;
}

// Driven Program
int main()
{
    string str = "3[b2[ca]]";
    cout << decode(str) << endl;
    return 0;
}
```

Java

```
// Java program to decode a string recursively
// encoded as count followed substring
```

```
import java.util.Stack;

class Test
{
    // Returns decoded string for 'str'
    static String decode(String str)
    {
        Stack<Integer> integerstack = new Stack<>();
        Stack<Character> stringstack = new Stack<>();
        String temp = "", result = "";

        // Traversing the string
        for (int i = 0; i < str.length(); i++)
        {
            int count = 0;

            // If number, convert it into number
            // and push it into integerstack.
            if (Character.isDigit(str.charAt(i)))
            {
                while (Character.isDigit(str.charAt(i)))
                {
                    count = count * 10 + str.charAt(i) - '0';
                    i++;
                }

                i--;
                integerstack.push(count);
            }

            // If closing bracket ']', pop element until
            // '[' opening bracket is not found in the
            // character stack.
            else if (str.charAt(i) == ']')
            {
                temp = "";
                count = 0;

                if (!integerstack.isEmpty())
                {
                    count = integerstack.peek();
                    integerstack.pop();
                }

                while (!stringstack.isEmpty() && stringstack.peek() != '[' )
                {
                    temp = stringstack.peek() + temp;
                }
            }
        }
    }
}
```

```
        stringstack.pop();
    }

    if (!stringstack.empty() && stringstack.peek() == '[')
        stringstack.pop();

    // Repeating the popped string 'temo' count
    // number of times.
    for (int j = 0; j < count; j++)
        result = result + temp;

    // Push it in the character stack.
    for (int j = 0; j < result.length(); j++)
        stringstack.push(result.charAt(j));

    result = "";
}

// If '[' opening bracket, push it into character stack.
else if (str.charAt(i) == '[')
{
    if (Character.isDigit(str.charAt(i-1)))
        stringstack.push(str.charAt(i));

    else
    {
        stringstack.push(str.charAt(i));
        integerstack.push(1);
    }
}

else
    stringstack.push(str.charAt(i));
}

// Pop all the elmenet, make a string and return.
while (!stringstack.isEmpty())
{
    result = stringstack.peek() + result;
    stringstack.pop();
}

return result;
}

// Driver method
public static void main(String args[])
{
```

```
        String str = "3[b2[ca]]";  
        System.out.println(decode(str));  
    }  
}
```

Output:

bcacabcacabcaca

Source

<https://www.geeksforgeeks.org/decode-string-recursively-encoded-count-followed-substring/>

Chapter 24

Delete array elements which are smaller than next or become smaller

Delete array elements which are smaller than next or become smaller - GeeksforGeeks

Given an array `arr[]` and a number `k`. The task is to delete `k` elements which are smaller than next element (i.e., we delete `arr[i]` if `arr[i] < arr[i+1]`) or become smaller than next because next element is deleted.

Examples:

Input : `arr[] = { 3, 100, 1 }`
 `k = 1`

Output : `100, 1`

Explanation : `arr[0] < arr[1]` means 3 is less than 100, so delete 3

Input : `arr[] = {20, 10, 25, 30, 40}`
 `k = 2`

Output : `25 30 40`

Explanation : First we delete 10 because it follows `arr[i] < arr[i+1]`. Then we delete 20 because 25 is moved next to it and it also starts following the condition.

Input : `arr[] = { 23, 45, 11, 77, 18}`
 `k = 3`

Output : `77, 18`

Explanation : We delete 23, 45 and 11 as they follow the condition `arr[i] < arr[i+1]`

Approach: [Stack](#) is used to solving this problem. First we push arr[0] in stack S and then initialize count as 0, then after traverse a loop from 1 to n and then we check that s.top() > arr[i] if condition is true then we pop the element from stack and increase the count if count == k then we stop the loop and then store the value of stack in another array and then print that array.

C++

```
// C++ program to delete elements from array.
#include <bits/stdc++.h>
using namespace std;

// Function for deleting k elements
void deleteElements(int arr[], int n, int k)
{
    // Create a stack and push arr[0]
    stack<int> s;
    s.push(arr[0]);

    int count = 0;

    // traversing a loop from i = 1 to n
    for (int i=1; i<n; i++) {

        // condition for deleting an element
        while (!s.empty() && s.top() < arr[i]
               && count < k) {
            s.pop();
            count++;
        }

        s.push(arr[i]);
    }

    // Putting elements of stack in a vector
    // from end to begin.
    int m = s.size();
    vector<int> v(m); // Size of vector is m
    while (!s.empty()) {

        // push element from stack to vector v
        v[--m] = s.top();
        s.pop();
    }

    // printing result
    for (auto x : v)
        cout << x << " ";
}
```

```
        cout << endl;
    }

    // Driver code
    int main()
    {
        int n = 5, k = 2;
        int arr[] = {20, 10, 25, 30, 40};
        deleteElements(arr, n, k);
        return 0;
    }
```

Python3

```
# Function to delete elements
def deleteElements(arr, n, k):

    # create an empty stack st
    st = []
    st.append(arr[0])

    # index to maintain the top
    # of the stack
    top = 0
    count = 0

    for i in range(1, n):

        # pop till the present element
        # is greater than stack's top
        # element
        while(len(st) != 0 and count < k
              and st[top] < arr[i]):
            st.pop()
            count += 1
            top -= 1

        st.append(arr[i])
        top += 1

    # print the remaining elements
    for i in range(0, len(st)):
        print(st[i], " ", end="")

# Driver code
k = 2
arr = [20, 10, 25, 30, 40]
deleteElements(arr, len(arr), k)
```


This code is contributed by himan085.

Output:

25 30 40

Improved By : [himan085](#)

Source

<https://www.geeksforgeeks.org/delete-array-elements-which-are-smaller-than-next-or-become-smaller/>

Chapter 25

Delete consecutive same words in a sequence

Delete consecutive same words in a sequence - GeeksforGeeks

Given a sequence of n strings, the task is to check if any two similar words come together then they destroy each other then print the number of words left in the sequence after this pairwise destruction.

Examples:

Input : ab aa aa bcd ab

Output : 3

As aa, aa destroys each other so, ab bcd ab is the new sequence.

Input : tom jerry jerry tom

Output : 0

As first both jerry will destroy each other.

Then sequence will be tom, tom they will also destroy each other. So, the final sequence doesn't contain any word.

Method 1:

- 1- Start traversing the sequence by storing it in vector.
 - a) Check if the current string is equal to next string if yes, erase both from the vector.
 - b) And check the same till last.
- 2- Return vector size.

C++

```
// C++ program to remove consecutive same words
#include<bits/stdc++.h>
using namespace std;

// Function to find the size of manipulated sequence
int removeConsecutiveSame(vector <string > v)
{
    int n = v.size();

    // Start traversing the sequence
    for (int i=0; i<n-1; )
    {
        // Compare the current string with next one
        // Erase both if equal
        if (v[i].compare(v[i+1]) == 0)
        {
            // Erase function delete the element and
            // also shifts other element that's why
            // i is not updated
            v.erase(v.begin()+i);
            v.erase(v.begin()+i);

            // Update i, as to check from previous
            // element again
            if (i > 0)
                i--;

            // Reduce sequence size
            n = n-2;
        }

        // Increment i, if not equal
        else
            i++;
    }

    // Return modified size
    return v.size();
}

// Driver code
int main()
{
    vector<string> v = { "tom", "jerry", "jerry", "tom"};
    cout << removeConsecutiveSame(v);
    return 0;
}
```

```
}
```

Java

```
// Java program to remove consecutive same words

import java.util.Vector;

class Test
{
    // Method to find the size of manipulated sequence
    static int removeConsecutiveSame(Vector <String > v)
    {
        int n = v.size();

        // Start traversing the sequence
        for (int i=0; i<n-1; )
        {
            // Compare the current string with next one
            // Erase both if equal
            if (v.get(i).equals(v.get(i+1)))
            {
                // Erase function delete the element and
                // also shifts other element that's why
                // i is not updated
                v.remove(i);
                v.remove(i);

                // Update i, as to check from previous
                // element again
                if (i > 0)
                    i--;

                // Reduce sequence size
                n = n-2;
            }

            // Increment i, if not equal
            else
                i++;
        }

        // Return modified size
        return v.size();
    }

    // Driver method
    public static void main(String[] args)
```

```
{
    Vector<String> v = new Vector<>();

    v.addElement("tom"); v.addElement("jerry");
    v.addElement("jerry");v.addElement("tom");

    System.out.println(removeConsecutiveSame(v));
}
}
```

Output:

0

Method 2:(Using Stack)

- 1- Start traversing the strings and push into stack.
 - a) Check if the current string is same as the string at the top of the stack
 - i) If yes, pop the string from top.
 - ii) Else push the current string.
- 2- Return stack size if the whole sequence is traversed.

C++

```
// C++ implementation of above method
#include<bits/stdc++.h>
using namespace std;

// Function to find the size of manipulated sequence
int removeConsecutiveSame(vector <string> v)
{
    stack<string> st;

    // Start traversing the sequence
    for (int i=0; i<v.size(); i++)
    {
        // Push the current string if the stack
        // is empty
        if (st.empty())
            st.push(v[i]);
        else
        {

```

```
        string str = st.top();

        // compare the current string with stack top
        // if equal, pop the top
        if (str.compare(v[i]) == 0)
            st.pop();

        // Otherwise push the current string
        else
            st.push(v[i]);
    }
}

// Return stack size
return st.size();
}

// Driver code
int main()
{
    vector<string> V = { "ab", "aa", "aa", "bcd", "ab"};
    cout << removeConsecutiveSame(V);
    return 0;
}
```

Java

```
// Java implementation of above method

import java.util.Stack;
import java.util.Vector;

class Test
{
    // Method to find the size of manipulated sequence
    static int removeConsecutiveSame(Vector <String> v)
    {
        Stack<String> st = new Stack<>();

        // Start traversing the sequence
        for (int i=0; i<v.size(); i++)
        {
            // Push the current string if the stack
            // is empty
            if (st.empty())
                st.push(v.get(i));
            else
            {

```

```
        String str = st.peek();

        // compare the current string with stack top
        // if equal, pop the top
        if (str.equals(v.get(i)))
            st.pop();

        // Otherwise push the current string
        else
            st.push(v.get(i));
    }
}

// Return stack size
return st.size();
}

// Driver method
public static void main(String[] args)
{
    Vector<String> v = new Vector<>();

    v.addElement("ab"); v.addElement("aa");
    v.addElement("aa");v.addElement("bcd");
    v.addElement("ab");

    System.out.println(removeConsecutiveSame(v));
}
}
```

Output:

3

Source

<https://www.geeksforgeeks.org/delete-consecutive-words-sequence/>

Chapter 26

Delete middle element of a stack

Delete middle element of a stack - GeeksforGeeks

Given a stack with push(), pop(), empty() operations, delete middle of it without using any additional data structure.

Input : Stack[] = [1, 2, 3, 4, 5]

Output : Stack[] = [1, 2, 4, 5]

Input : Stack[] = [1, 2, 3, 4, 5, 6]

Output : Stack[] = [1, 2, 4, 5, 6]

The idea is to use recursive calls. We first remove all items one by one, then we recur. After recursive calls, we push all items back except the middle item.

C++

```
// C++ code to delete middle of a stack
// without using additional data structure.
#include<bits/stdc++.h>
using namespace std;

// Deletes middle of stack of size
// n. Curr is current item number
void deleteMid(stack<char> &st, int n,
               int curr=0)
{
    // If stack is empty or all items
    // are traversed
    if (st.empty() || curr == n)
        return;
```



```
// Remove current item
int x = st.top();
st.pop();

// Remove other items
deleteMid(st, n, curr+1);

// Put all items back except middle
if (curr != n/2)
    st.push(x);
}

//Driver function to test above functions
int main()
{
    stack<char> st;

    //push elements into the stack
    st.push('1');
    st.push('2');
    st.push('3');
    st.push('4');
    st.push('5');
    st.push('6');
    st.push('7');

    deleteMid(st, st.size());

    // Printing stack after deletion
    // of middle.
    while (!st.empty())
    {
        char p=st.top();
        st.pop();
        cout << p << " ";
    }
    return 0;
}
```

Java

```
// Java code to delete middle of a stack
// without using additional data structure.
import java.io.*;
import java.util.*;

public class GFG {
```

```
// Deletes middle of stack of size
// n. Curr is current item number
static void deleteMid(Stack<Character> st,
                      int n, int curr)
{
    // If stack is empty or all items
    // are traversed
    if (st.empty() || curr == n)
        return;

    // Remove current item
    char x = st.pop();

    // Remove other items
    deleteMid(st, n, curr+1);

    // Put all items back except middle
    if (curr != n/2)
        st.push(x);
}

// Driver function to test above functions
public static void main(String args[])
{
    Stack<Character> st =
        new Stack<Character>();

    // push elements into the stack
    st.push('1');
    st.push('2');
    st.push('3');
    st.push('4');
    st.push('5');
    st.push('6');
    st.push('7');

    deleteMid(st, st.size(), 0);

    // Printing stack after deletion
    // of middle.
    while (!st.empty())
    {
        char p=st.pop();
        System.out.print(p + " ");
    }
}
```

```
}

// This code is contributed by
// Manish Shaw (manishshaw1)

Python3

# Python3 code to delete middle of a stack
# without using additional data structure.

# Deletes middle of stack of size
# n. Curr is current item number
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)

def deleteMid(st, n, curr) :

    # If stack is empty or all items
    # are traversed
    if (st.isEmpty() or curr == n) :
        return

    # Remove current item
    x = st.peek()
    st.pop()

    # Remove other items
    deleteMid(st, n, curr+1)

    # Put all items back except middle
    if (curr != int(n/2)) :
        st.push(x)
```

```
# Driver function to test above functions
st = Stack()

# push elements into the stack
st.push('1')
st.push('2')
st.push('3')
st.push('4')
st.push('5')
st.push('6')
st.push('7')

deleteMid(st, st.size(), 0)

# Printing stack after deletion
# of middle.
while (st.isEmpty() == False) :
    p = st.peek()
    st.pop()
    print (str(p) + " ", end="")

# This code is contributed by
# Manish Shaw (manishshaw1)
```

C#

```
// C# code to delete middle of a stack
// without using additional data structure.
using System;
using System.Collections.Generic;

class GFG {

    // Deletes middle of stack of size
    // n. Curr is current item number
    static void deleteMid(Stack<char> st,
                          int n,
                          int curr = 0)
    {

        // If stack is empty or all
        // items are traversed
        if (st.Count == 0 || curr == n)
            return;

        // Remove current item
        char x = st.Peek();
```

```
        st.Pop();

        // Remove other items
        deleteMid(st, n, curr+1);

        // Put all items
        // back except middle
        if (curr != n / 2)
            st.Push(x);
    }

    // Driver Code
    public static void Main()
    {
        Stack<char> st = new Stack<char>();

        // push elements into the stack
        st.Push('1');
        st.Push('2');
        st.Push('3');
        st.Push('4');
        st.Push('5');
        st.Push('6');
        st.Push('7');

        deleteMid(st, st.Count);

        // Printing stack after
        // deletion of middle.
        while (st.Count != 0)
        {
            char p=st.Peek();
            st.Pop();
            Console.Write(p + " ");
        }
    }
}

// This code is contributed by
// Manish Shaw (manishshaw1)
```

Output:

7 6 5 3 2 1

Improved By : [manishshaw1](#)

Source

<https://www.geeksforgeeks.org/delete-middle-element-stack/>

Chapter 27

Design a stack that supports getMin() in $O(1)$ time and $O(1)$ extra space

Design a stack that supports getMin() in $O(1)$ time and $O(1)$ extra space - GeeksforGeeks

Question: Design a Data Structure SpecialStack that supports all the stack operations like push(), pop(), isEmpty(), isFull() and an additional operation getMin() which should return minimum element from the SpecialStack. All these operations of SpecialStack must be $O(1)$. To implement SpecialStack, you should only use standard Stack data structure and no other data structure like arrays, list, .. etc.

Example:

Consider the following SpecialStack

```
16 --> TOP
15
29
19
18
```

When getMin() is called it should return 15,
which is the minimum element in the current stack.

If we do pop two times on stack, the stack becomes

```
29 --> TOP
19
18
```

When getMin() is called, it should return 18
which is the minimum in the current stack.

An approach that uses $O(1)$ time and $O(n)$ extra space is discussed [here](#).

In this article, a new approach is discussed that supports minimum with $O(1)$ extra space. We define a variable **minEle** that stores the current minimum element in the stack. Now the interesting part is, how to handle the case when minimum element is removed. To handle this, we push “ $2x - \text{minEle}$ ” into the stack instead of x so that previous minimum element can be retrieved using current minEle and its value stored in stack. Below are detailed steps and explanation of working.

Push(x) : Inserts x at the top of stack.

- If stack is empty, insert x into the stack and make minEle equal to x .
- If stack is not empty, compare x with minEle. Two cases arise:
 - If x is greater than or equal to minEle, simply insert x .
 - If x is less than minEle, insert $(2*x - \text{minEle})$ into the stack and make minEle equal to x . For example, let previous minEle was 3. Now we want to insert 2. We update minEle as 2 and insert $2*2 - 3 = 1$ into the stack.

Pop() : Removes an element from top of stack.

- Remove element from top. Let the removed element be y . Two cases arise:
 - If y is greater than or equal to minEle, the minimum element in the stack is still minEle.
 - If y is less than minEle, the minimum element now becomes $(2*\text{minEle} - y)$, so update $(\text{minEle} = 2*\text{minEle} - y)$. This is where we retrieve previous minimum from current minimum and its value in stack. For example, let the element to be removed be 1 and minEle be 2. We remove 1 and update minEle as $2*2 - 1 = 3$.

Important Points:

- Stack doesn't hold actual value of an element if it is minimum so far.
- Actual minimum element is always stored in minEle

Illustration

Push(x)

Number Inserted	Present Stack	minEle
3	3	3
5	5 3	3
2	1 5 3	2
1	0 1 5 3	1
1	1 0 1 5 3	1
-1	-3 1 0 1 5 3	-1

- Number to be Inserted: 3, Stack is empty, so insert 3 into stack and minEle = 3.
- Number to be Inserted: 5, Stack is not empty, $5 > \text{minEle}$, insert 5 into stack and minEle = 3.
- Number to be Inserted: 2, Stack is not empty, $2 < \text{minEle}$, insert $(2*2-3 = 1)$ into stack and minEle = 2.
- Number to be Inserted: 1, Stack is not empty, $1 < \text{minEle}$, insert $(2*1-2 = 0)$ into stack and minEle = 1.
- Number to be Inserted: 1, Stack is not empty, $1 = \text{minEle}$, insert 1 into stack and minEle = 1.
- Number to be Inserted: -1, Stack is not empty, $-1 < \text{minEle}$, insert $(2*-1 - 1 = -3)$ into stack and minEle = -1.

Pop()

Number Removed	Original Number	Present Stack	minEle
-	-	-3 1 0 1 5 3	-1
-3	-1	1 0 1 5 3	1
1	1	0 1 5 3	1
0	1	1 5 3	2
1	2	5 3	3
5	5	3	3

- Initially the minimum element minEle in the stack is -1.
- Number removed: -3, Since -3 is less than the minimum element the original number being removed is minEle which is -1, and the new minEle = $2 * -1 - (-3) = 1$
- Number removed: 1, $1 == \text{minEle}$, so number removed is 1 and minEle is still equal to 1.
- Number removed: 0, $0 < \text{minEle}$, original number is minEle which is 1 and new minEle = $2 * 1 - 0 = 2$.
- Number removed: 1, $1 < \text{minEle}$, original number is minEle which is 2 and new minEle = $2 * 2 - 1 = 3$.
- Number removed: 5, $5 > \text{minEle}$, original number is 5 and minEle is still 3

C++

```
// C++ program to implement a stack that supports
// getMinimum() in  $O(1)$  time and  $O(1)$  extra space.
#include <bits/stdc++.h>
using namespace std;

// A user defined stack that supports getMin() in
// addition to push() and pop()
struct MyStack
{
    stack<int> s;
    int minEle;

    // Prints minimum element of MyStack
    void getMin()
    {
        if (s.empty())
```

```
        cout << "Stack is empty\n";

        // variable minEle stores the minimum element
        // in the stack.
        else
            cout << "Minimum Element in the stack is: "
                << minEle << "\n";
    }

    // Prints top element of MyStack
    void peek()
    {
        if (s.empty())
        {
            cout << "Stack is empty ";
            return;
        }

        int t = s.top(); // Top element.

        cout << "Top Most Element is: ";

        // If t < minEle means minEle stores
        // value of t.
        (t < minEle)? cout << minEle: cout << t;
    }

    // Remove the top element from MyStack
    void pop()
    {
        if (s.empty())
        {
            cout << "Stack is empty\n";
            return;
        }

        cout << "Top Most Element Removed: ";
        int t = s.top();
        s.pop();

        // Minimum will change as the minimum element
        // of the stack is being removed.
        if (t < minEle)
        {
            cout << minEle << "\n";
            minEle = 2*minEle - t;
        }
    }
}
```

```
        else
            cout << t << "\n";
    }

    // Removes top element from MyStack
    void push(int x)
    {
        // Insert new number into the stack
        if (s.empty())
        {
            minEle = x;
            s.push(x);
            cout << "Number Inserted: " << x << "\n";
            return;
        }

        // If new number is less than minEle
        if (x < minEle)
        {
            s.push(2*x - minEle);
            minEle = x;
        }

        else
            s.push(x);

        cout << "Number Inserted: " << x << "\n";
    }
};

// Driver Code
int main()
{
    MyStack s;
    s.push(3);
    s.push(5);
    s.getMin();
    s.push(2);
    s.push(1);
    s.getMin();
    s.pop();
    s.getMin();
    s.pop();
    s.peak();

    return 0;
}
```

Java

```
// Java program to implement a stack that supports
// getMinimum() in  $O(1)$  time and  $O(1)$  extra space.
import java.util.*;

// A user defined stack that supports getMin() in
// addition to push() and pop()
class MyStack
{
    Stack<Integer> s;
    Integer minEle;

    // Constructor
    MyStack() { s = new Stack<Integer>(); }

    // Prints minimum element of MyStack
    void getMin()
    {
        // Get the minimum number in the entire stack
        if (s.isEmpty())
            System.out.println("Stack is empty");

        // variable minEle stores the minimum element
        // in the stack.
        else
            System.out.println("Minimum Element in the " +
                               " stack is: " + minEle);
    }

    // prints top element of MyStack
    void peek()
    {
        if (s.isEmpty())
        {
            System.out.println("Stack is empty ");
            return;
        }

        Integer t = s.peek(); // Top element.

        System.out.print("Top Most Element is: ");

        // If t < minEle means minEle stores
        // value of t.
        if (t < minEle)
            System.out.println(minEle);
        else
```

```
        System.out.println(t);
    }

    // Removes the top element from MyStack
    void pop()
    {
        if (s.isEmpty())
        {
            System.out.println("Stack is empty");
            return;
        }

        System.out.print("Top Most Element Removed: ");
        Integer t = s.pop();

        // Minimum will change as the minimum element
        // of the stack is being removed.
        if (t < minEle)
        {
            System.out.println(minEle);
            minEle = 2*minEle - t;
        }

        else
            System.out.println(t);
    }

    // Insert new number into MyStack
    void push(Integer x)
    {
        if (s.isEmpty())
        {
            minEle = x;
            s.push(x);
            System.out.println("Number Inserted: " + x);
            return;
        }

        // If new number is less than original minEle
        if (x < minEle)
        {
            s.push(2*x - minEle);
            minEle = x;
        }

        else
            s.push(x);
    }
}
```

```
        System.out.println("Number Inserted: " + x);
    }
};

// Driver Code
public class Main
{
    public static void main(String[] args)
    {
        MyStack s = new MyStack();
        s.push(3);
        s.push(5);
        s.getMin();
        s.push(2);
        s.push(1);
        s.getMin();
        s.pop();
        s.getMin();
        s.pop();
        s.peek();
    }
}
```

Output:

```
Number Inserted: 3
Number Inserted: 5
Minimum Element in the stack is: 3
Number Inserted: 2
Number Inserted: 1
Minimum Element in the stack is: 1
Top Most Element Removed: 1
Minimum Element in the stack is: 2
Top Most Element Removed: 2
Top Most Element is: 5
```

How does this approach work?

When element to be inserted is less than minEle, we insert " $2x - \text{minEle}$ ". The important thing to notes is, $2x - \text{minEle}$ will always be less than x (proved below), i.e., new minEle and while popping out this element we will see that something unusual has happened as the popped element is less than the minEle. So we will be updating minEle.

How $2*x - \text{minEle}$ is less than x in push()?
 $x < \text{minEle}$ which means $x - \text{minEle} < 0$

// Adding x on both sides

$x - \text{minEle} + x < 0 + x$

$2*x - \text{minEle} < x$

We can conclude $2*x - \text{minEle} < \text{new minEle}$

While popping out, if we find the element(y) less than the current minEle, we find the new minEle = $2*\text{minEle} - y$.

How previous minimum element, prevMinEle is, $2*\text{minEle} - y$ in pop() is y the popped element?

```
// We pushed y as 2x - prevMinEle. Here
// prevMinEle is minEle before y was inserted
y = 2*x - prevMinEle

// Value of minEle was made equal to x
minEle = x .

new minEle = 2 * minEle - y
            = 2*x - (2*x - prevMinEle)
            = prevMinEle // This is what we wanted
```

Exercise:

Similar approach can be used to find the maximum element as well. Implement a stack that supports getMax() in O(1) time and constant extra space.

Source

<https://www.geeksforgeeks.org/design-a-stack-that-supports-getmin-in-o1-time-and-o1-extra-space/>

Chapter 28

Design a stack to retrieve original elements and return the minimum element in $O(1)$ time and $O(1)$ space

Design a stack to retrieve original elements and return the minimum element in $O(1)$ time and $O(1)$ space - GeeksforGeeks

Our task is to design a Data Structure SpecialStack that supports all the stack operations like **push()**, **pop()**, **isEmpty()**, **isFull()** and an additional operation **getMin()** which should return minimum element from the SpecialStack. All these operations of SpecialStack must be performed with time complexity $O(1)$. To implement SpecialStack, you should only use standard Stack data structure and no other data structure like arrays, list etc.

Example:

Consider the following Special-Stack

```
16 --> TOP
15
29
19
18
```

When **getMin()** is called it should return 15, which is the minimum element in the current stack.

If we do pop two times on stack, the stack becomes

```
29 --> TOP
19
```

18

When `getMin()` is called, it should return 18 which is the minimum in the current stack.

An **approach** that uses $O(1)$ time and $O(1)$ extra space is discussed [here](#). However, in the previous article the original elements are not recovered. Only the minimum element is returned at any given point of time.

In this article, the previous approach is modified so that original elements can also be retrieved during a **pop()** operation.

Approach:

Consider a variable **minimum** in which we store the minimum element in the stack. Now, what if we pop the minimum element from the stack? How do we update the minimum variable to the next minimum value? One solution is to maintain another stack in sorted order so that the smallest element is always on the top. However, this is an $O(n)$ space approach.

To achieve this in **$O(1)$** space, we need a way to store the current value of an element and the next minimum value in the same node. This can be done by applying simple mathematics:

```
new_value = 2*current_value - minimum
```

We push this `new_value` into the stack instead of `current_value`. To retrieve `current_value` and next minimum from `new_value`:

```
current_value = (new_value + minimum)/2  
minimum = new_value - 2*current
```

When the operation **Push(x)** is done, we follow the given below algorithm:

1. If stack is empty
 - (a) insert x into the stack
 - (b) make minimum equal to x.
2. If stack is not empty
 - (a) if x is less than minimum
 - i. set temp equal to $2*x - \text{minimum}$
 - ii. set minimum equal to x
 - iii. set x equal to temp
 - (b) insert x into stack

When the operation **Pop(x)** is done, we follow the given below algorithm:

1. If stack is not empty

- (a) set x equal to topmost element
- (b) if x is less than minimum
 - i. set minimum equal to $2 * \text{minimum} - x$
 - ii. set x equal to $(x + \text{minimum}) / 2$
- (c) return x

When `getMin()` is called, we return the element stored in variable, *minimum*:

```
// Java program to retrieve original elements of the
// from a Stack which returns the minimum element
// in O(1) time and O(1) space

class Stack {
    Node top;

    // Stores minimum element of the stack
    int minimum;

    // Function to push an element
    void push(Node node)
    {
        int current = node.getData();
        if (top == null) {
            top = node;
            minimum = current;
        }
        else {
            if (current < minimum) {
                node.setData(2 * current - minimum);
                minimum = current;
            }
            node.setNext(top);
            top = node;
        }
    }

    // Retrieves topmost element
    Node pop()
    {
        Node node = top;
        if (node != null) {
            int current = node.getData();
            if (current < minimum) {
                minimum = 2 * minimum - current;
                node.setData((current + minimum) / 2);
            }
            top = top.getNext();
        }
    }
}
```

```
    }
    return node;
}

// Retrieves topmost element without
// popping it from the stack
Node peek()
{
    Node node = null;
    if (top != null) {
        node = new Node();
        int current = top.getData();
        node.setData(current < minimum ? minimum : current);
    }
    return node;
}

// Function to print all elements in the stack
void printAll()
{
    Node ptr = top;
    int min = minimum;
    if (ptr != null) { // if stack is not empty
        while (true) {
            int val = ptr.getData();
            if (val < min) {
                min = 2 * min - val;
                val = (val + min) / 2;
            }
            System.out.print(val + " ");
            ptr = ptr.getNext();
            if (ptr == null)
                break;
        }
        System.out.println();
    }
    else
        System.out.println("Empty!");
}

// Returns minimum of Stack
int getMin()
{
    return minimum;
}

boolean isEmpty()
{

```

```
        return top == null;
    }
}

// Node class which contains data
// and pointer to next node
class Node {
    int data;
    Node next;
    Node()
    {
        data = -1;
        next = null;
    }

    Node(int d)
    {
        data = d;
        next = null;
    }

    void setData(int data)
    {
        this.data = data;
    }

    void setNext(Node next)
    {
        this.next = next;
    }

    Node getNext()
    {
        return next;
    }

    int getData()
    {
        return data;
    }
}

// Driver Code
public class Main {
    public static void main(String[] args)
    {
        // Create a new stack
        Stack stack = new Stack();
    }
}
```

```
Node node;

// Push the element into the stack
stack.push(new Node(5));
stack.push(new Node(3));
stack.push(new Node(4));

// Calls the method to print the stack
System.out.println("Elements in the stack are:");
stack.printAll();

// Print current minimum element if stack is
// not empty
System.out.println(stack.isEmpty() ? "\nEmpty Stack!" :
    "\nMinimum: " + stack.getMin());

// Push new elements into the stack
stack.push(new Node(1));
stack.push(new Node(2));

// Printing the stack
System.out.println("\nStack after adding new elements:");
stack.printAll();

// Print current minimum element if stack is
// not empty
System.out.println(stack.isEmpty() ? "\nEmpty Stack!" :
    "\nMinimum: " + stack.getMin());

// Pop elements from the stack
node = stack.pop();
System.out.println("\nElement Popped: "
    + (node == null ? "Empty!" : node.getData()));

node = stack.pop();
System.out.println("Element Popped: "
    + (node == null ? "Empty!" : node.getData()));

// Printing stack after popping elements
System.out.println("\nStack after removing top two elements:");
stack.printAll();

// Printing current Minimum element in the stack
System.out.println(stack.isEmpty() ? "\nEmpty Stack!" :
    "\nMinimum: " + stack.getMin());

// Printing top element of the stack
node = stack.peek();
```

```
        System.out.println("\nTop: " + (node == null ?
                                "\nEmpty!" : node.getData()));
    }
}
```

Output:

Elements in the stack are:

4 3 5

Minimum: 3

Stack after adding new elements:

2 1 4 3 5

Minimum: 1

Element Popped: 2

Element Popped: 1

Stack after removing top two elements:

4 3 5

Minimum: 3

Top: 4

Source

<https://www.geeksforgeeks.org/design-a-stack-to-retrieve-original-elements-and-return-the-minimum-element-in-o1-time-and-o1-space/>

Chapter 29

Design a stack with operations on middle element

Design a stack with operations on middle element - GeeksforGeeks

How to implement a stack which will support following operations in **O(1) time complexity**?

- 1) push() which adds an element to the top of stack.
- 2) pop() which removes an element from top of stack.
- 3) findMiddle() which will return middle element of the stack.
- 4) deleteMiddle() which will delete the middle element.

Push and pop are standard stack operations.

The important question is, whether to use a linked list or array for implementation of stack?

Please note that, we need to find and delete middle element. Deleting an element from middle is not O(1) for array. Also, we may need to move the middle pointer up when we push an element and move down when we pop(). In singly linked list, moving middle pointer in both directions is not possible.

The idea is to use Doubly Linked List (DLL). We can delete middle element in O(1) time by maintaining mid pointer. We can move mid pointer in both directions using previous and next pointers.

Following is implementation of push(), pop() and findMiddle() operations. Implementation of deleteMiddle() is left as an exercise. If there are even elements in stack, findMiddle() returns the first middle element. For example, if stack contains {1, 2, 3, 4}, then findMiddle() would return 2.

C/C++

```
/* Program to implement a stack that supports findMiddle() and deleteMiddle
   in O(1) time */
#include <stdio.h>
#include <stdlib.h>
```



```
/* A Doubly Linked List Node */
struct DLLNode
{
    struct DLLNode *prev;
    int data;
    struct DLLNode *next;
};

/* Representation of the stack data structure that supports findMiddle()
   in O(1) time. The Stack is implemented using Doubly Linked List. It
   maintains pointer to head node, pointer to middle node and count of
   nodes */
struct myStack
{
    struct DLLNode *head;
    struct DLLNode *mid;
    int count;
};

/* Function to create the stack data structure */
struct myStack *createMyStack()
{
    struct myStack *ms =
        (struct myStack*) malloc(sizeof(struct myStack));
    ms->count = 0;
    return ms;
};

/* Function to push an element to the stack */
void push(struct myStack *ms, int new_data)
{
    /* allocate DLLNode and put in data */
    struct DLLNode* new_DLLNode =
        (struct DLLNode*) malloc(sizeof(struct DLLNode));
    new_DLLNode->data = new_data;

    /* Since we are adding at the begining,
       prev is always NULL */
    new_DLLNode->prev = NULL;

    /* link the old list off the new DLLNode */
    new_DLLNode->next = ms->head;

    /* Increment count of items in stack */
    ms->count += 1;

    /* Change mid pointer in two cases
```

```
    1) Linked List is empty
    2) Number of nodes in linked list is odd */
if (ms->count == 1)
{
    ms->mid = new_DLLNode;
}
else
{
    ms->head->prev = new_DLLNode;

    if (ms->count & 1) // Update mid if ms->count is odd
        ms->mid = ms->mid->prev;
}

/* move head to point to the new DLLNode */
ms->head = new_DLLNode;
}

/* Function to pop an element from stack */
int pop(struct myStack *ms)
{
    /* Stack underflow */
    if (ms->count == 0)
    {
        printf("Stack is empty\n");
        return -1;
    }

    struct DLLNode *head = ms->head;
    int item = head->data;
    ms->head = head->next;

    // If linked list doesn't become empty, update prev
    // of new head as NULL
    if (ms->head != NULL)
        ms->head->prev = NULL;

    ms->count -= 1;

    // update the mid pointer when we have even number of
    // elements in the stack, i.e move down the mid pointer.
    if (!(ms->count & 1))
        ms->mid = ms->mid->next;

    free(head);

    return item;
}
```

```
// Function for finding middle of the stack
int findMiddle(struct myStack *ms)
{
    if (ms->count == 0)
    {
        printf("Stack is empty now\n");
        return -1;
    }

    return ms->mid->data;
}

// Driver program to test functions of myStack
int main()
{
    /* Let us create a stack using push() operation*/
    struct myStack *ms = createMyStack();
    push(ms, 11);
    push(ms, 22);
    push(ms, 33);
    push(ms, 44);
    push(ms, 55);
    push(ms, 66);
    push(ms, 77);

    printf("Item popped is %d\n", pop(ms));
    printf("Item popped is %d\n", pop(ms));
    printf("Middle Element is %d\n", findMiddle(ms));
    return 0;
}
```

Java

```
/* Java Program to implement a stack that supports findMiddle() and deleteMiddle
in O(1) time */

public class GFG
{
    /* A Doubly Linked List Node */
    class DLLNode
    {
        DLLNode prev;
        int data;
        DLLNode next;
        DLLNode(int d){data=d;}
    }
}
```

```
/* Representation of the stack data structure that supports findMiddle()
   in O(1) time. The Stack is implemented using Doubly Linked List. It
   maintains pointer to head node, pointer to middle node and count of
   nodes */
class myStack
{
    DLLNode head;
    DLLNode mid;
    int count;
}

/* Function to create the stack data structure */
myStack createMyStack()
{
    myStack ms = new myStack();
    ms.count = 0;
    return ms;
}

/* Function to push an element to the stack */
void push(myStack ms, int new_data)
{
    /* allocate DLLNode and put in data */
    DLLNode new_DLLNode = new DLLNode(new_data);

    /* Since we are adding at the beginning,
       prev is always NULL */
    new_DLLNode.prev = null;

    /* link the old list off the new DLLNode */
    new_DLLNode.next = ms.head;

    /* Increment count of items in stack */
    ms.count += 1;

    /* Change mid pointer in two cases
       1) Linked List is empty
       2) Number of nodes in linked list is odd */
    if(ms.count == 1)
    {
        ms.mid=new_DLLNode;
    }
    else
    {

```

```
        ms.head.prev = new_DLLNode;

        if((ms.count % 2) != 0) // Update mid if ms->count is odd
            ms.mid=ms.mid.prev;
    }

    /* move head to point to the new DLLNode */
    ms.head = new_DLLNode;
}

/* Function to pop an element from stack */
int pop(myStack ms)
{
    /* Stack underflow */
    if(ms.count == 0)
    {
        System.out.println("Stack is empty");
        return -1;
    }

    DLLNode head = ms.head;
    int item = head.data;
    ms.head = head.next;

    // If linked list doesn't become empty, update prev
    // of new head as NULL
    if(ms.head != null)
        ms.head.prev = null;

    ms.count -= 1;

    // update the mid pointer when we have even number of
    // elements in the stack, i.e move down the mid pointer.
    if(ms.count % 2 == 0)
        ms.mid=ms.mid.next;

    return item;
}

// Function for finding middle of the stack
int findMiddle(myStack ms)
{
    if(ms.count == 0)
    {
        System.out.println("Stack is empty now");
        return -1;
    }
}
```

```
        return ms.mid.data;
    }

    // Driver program to test functions of myStack
    public static void main(String args[])
    {
        GFG ob = new GFG();
        myStack ms = ob.createMyStack();
        ob.push(ms, 11);
        ob.push(ms, 22);
        ob.push(ms, 33);
        ob.push(ms, 44);
        ob.push(ms, 55);
        ob.push(ms, 66);
        ob.push(ms, 77);

        System.out.println("Item popped is " + ob.pop(ms));
        System.out.println("Item popped is " + ob.pop(ms));
        System.out.println("Middle Element is " + ob.findMiddle(ms));
    }
}

// This code is contributed by Sumit Ghosh
```

Output:

```
Item popped is 77
Item popped is 66
Middle Element is 33
```

This article is contributed by [Chandra Prakash](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/design-a-stack-with-find-middle-operation/>

Chapter 30

Design and Implement Special Stack Data Structure | Added Space Optimized Version

Design and Implement Special Stack Data Structure | Added Space Optimized Version - GeeksforGeeks

Question: Design a Data Structure SpecialStack that supports all the stack operations like push(), pop(), isEmpty(), isFull() and an additional operation getMin() which should return minimum element from the SpecialStack. All these operations of SpecialStack must be O(1). To implement SpecialStack, you should only use standard Stack data structure and no other data structure like arrays, list, .. etc.

Example:

Consider the following SpecialStack

```
16 --> TOP
15
29
19
18
```

When getMin() is called it should return 15, which is the minimum element in the current stack.

If we do pop two times on stack, the stack becomes

```
29 --> TOP
19
18
```

When getMin() is called, it should return 18 which is the minimum

in the current stack.

Solution: Use two stacks: one to store actual stack elements and other as an auxiliary stack to store minimum values. The idea is to do push() and pop() operations in such a way that the top of auxiliary stack is always the minimum. Let us see how push() and pop() operations work.

Push(int x) // inserts an element x to Special Stack

- 1) push x to the first stack (the stack with actual elements)
- 2) compare x with the top element of the second stack (the auxiliary stack). Let the top element be y.
.....a) If x is smaller than y then push x to the auxiliary stack.
.....b) If x is greater than y then push y to the auxiliary stack.

int Pop() // removes an element from Special Stack and return the removed element

- 1) pop the top element from the auxiliary stack.
- 2) pop the top element from the actual stack and return it.

The step 1 is necessary to make sure that the auxiliary stack is also updated for future operations.

int getMin() // returns the minimum element from Special Stack

- 1) Return the top element of auxiliary stack.

We can see that **all above operations are O(1)**.

Let us see an example. Let us assume that both stacks are initially empty and 18, 19, 29, 15 and 16 are inserted to the SpecialStack.

When we insert 18, both stacks change to following.

Actual Stack

18

Following is implementation for SpecialStack class. In the below implementation, SpecialStack is

C++

```
#include<iostream>
#include<stdlib.h>

using namespace std;

/* A simple stack class with basic stack functionalities */
class Stack
{
private:
    static const int max = 100;
    int arr[max];
    int top;
```



```
public:
    Stack() { top = -1; }
    bool isEmpty();
    bool isFull();
    int pop();
    void push(int x);
};

/* Stack's member method to check if the stack is iempty */
bool Stack::isEmpty()
{
    if(top == -1)
        return true;
    return false;
}

/* Stack's member method to check if the stack is full */
bool Stack::isFull()
{
    if(top == max - 1)
        return true;
    return false;
}

/* Stack's member method to remove an element from it */
int Stack::pop()
{
    if(isEmpty())
    {
        cout<<"Stack Underflow";
        abort();
    }
    int x = arr[top];
    top--;
    return x;
}

/* Stack's member method to insert an element to it */
void Stack::push(int x)
{
    if(isFull())
    {
        cout<<"Stack Overflow";
        abort();
    }
    top++;
    arr[top] = x;
}
```

```
/* A class that supports all the stack operations and one additional
operation getMin() that returns the minimum element from stack at
any time. This class inherits from the stack class and uses an
auxiliary stack that holds minimum elements */
class SpecialStack: public Stack
{
    Stack min;
public:
    int pop();
    void push(int x);
    int getMin();
};

/* SpecialStack's member method to insert an element to it. This method
makes sure that the min stack is also updated with appropriate minimum
values */
void SpecialStack::push(int x)
{
    if(isEmpty()==true)
    {
        Stack::push(x);
        min.push(x);
    }
    else
    {
        Stack::push(x);
        int y = min.pop();
        min.push(y);
        if( x < y )
            min.push(x);
        else
            min.push(y);
    }
}

/* SpecialStack's member method to remove an element from it. This method
removes top element from min stack also. */
int SpecialStack::pop()
{
    int x = Stack::pop();
    min.pop();
    return x;
}

/* SpecialStack's member method to get minimum element from it. */
int SpecialStack::getMin()
{
```

```
        int x = min.pop();
        min.push(x);
        return x;
    }

    /* Driver program to test SpecialStack methods */
    int main()
    {
        SpecialStack s;
        s.push(10);
        s.push(20);
        s.push(30);
        cout<<s.getMin()<<endl;
        s.push(5);
        cout<<s.getMin();
        return 0;
    }
```

Java

```
//Java implementation of SpecialStack
// Note : here we use Stack class for Stack implementation

import java.util.Stack;

/* A class that supports all the stack operations and one additional
operation getMin() that returns the minimum element from stack at
any time. This class inherits from the stack class and uses an
auxiliary stack that holds minimum elements */

class SpecialStack extends Stack<Integer>
{
    Stack<Integer> min = new Stack<>();

    /* SpecialStack's member method to insert an element to it. This method
    makes sure that the min stack is also updated with appropriate minimum
    values */
    void push(int x)
    {
        if(isEmpty() == true)
        {
            super.push(x);
            min.push(x);
        }
        else
        {
            super.push(x);
            int y = min.pop();
        }
    }
}
```

```
        min.push(y);
        if(x < y)
            min.push(x);
        else
            min.push(y);
    }
}

/* SpecialStack's member method to insert an element to it. This method
makes sure that the min stack is also updated with appropriate minimum
values */
public Integer pop()
{
    int x = super.pop();
    min.pop();
    return x;
}

/* SpecialStack's member method to get minimum element from it. */
int getMin()
{
    int x = min.pop();
    min.push(x);
    return x;
}

/* Driver program to test SpecialStack methods */
public static void main(String[] args)
{
    SpecialStack s = new SpecialStack();
    s.push(10);
    s.push(20);
    s.push(30);
    System.out.println(s.getMin());
    s.push(5);
    System.out.println(s.getMin());
}
}
// This code is contributed by Sumit Ghosh
```

Output:

10
5

Space Optimized Version

The above approach can be optimized. We can limit the number of elements in auxiliary stack. We can push only when the incoming element of main stack is smaller than or equal to top of auxiliary stack. Similarly during pop, if the pop off element equal to top of auxiliary

stack, remove the top element of auxiliary stack. Following is modified implementation of push() and pop().

C++

```
/* SpecialStack's member method to insert an element to it. This method
   makes sure that the min stack is also updated with appropriate minimum
   values */
void SpecialStack::push(int x)
{
    if(isEmpty()==true)
    {
        Stack::push(x);
        min.push(x);
    }
    else
    {
        Stack::push(x);
        int y = min.pop();
        min.push(y);

        /* push only when the incoming element of main stack is smaller
        than or equal to top of auxiliary stack */
        if( x <= y )
            min.push(x);
    }
}

/* SpecialStack's member method to remove an element from it. This method
   removes top element from min stack also. */
int SpecialStack::pop()
{
    int x = Stack::pop();
    int y = min.pop();

    /* Push the popped element y back only if it is not equal to x */
    if ( y != x )
        min.push(y);

    return x;
}
```

Java

```
/* SpecialStack's member method to insert an element to it. This method
   makes sure that the min stack is also updated with appropriate minimum
   values */
```

```
void push(int x)
{
    if(isEmpty() == true)
    {
        super.push(x);
        min.push(x);
    }
    else
    {
        super.push(x);
        int y = min.pop();
        min.push(y);

        /* push only when the incoming element of main stack is smaller
        than or equal to top of auxiliary stack */
        if( x <= y )
            min.push(x);
    }
}

/* SpecialStack's member method to remove an element from it. This method
removes top element from min stack also. */
public Integer pop()
{
    int x = super.pop();
    int y = min.pop();

    /* Push the popped element y back only if it is not equal to x */
    if ( y != x )
        min.push(y);
    return x;
}

// This code is contributed by Sumit Ghosh
```

Design a stack that supports getMin() in O(1) time and O(1) extra space

Thanks to @Venki, @swarup and @Jing Huang for their inputs.

Source

<https://www.geeksforgeeks.org/design-and-implement-special-stack-data-structure/>

Chapter 31

Evaluation of Prefix Expressions

Evaluation of Prefix Expressions - GeeksforGeeks

Prefix and Postfix expressions can be evaluated faster than an infix expression. This is because we don't need to process any brackets or follow operator precedence rule. In postfix and prefix expressions which ever operator comes before will be evaluated first, irrespective of its priority. Also, there are no brackets in these expressions. As long as we can guarantee that a valid prefix or postfix expression is used, it can be evaluated with correctness.

We can convert [infix to postfix](#) and can convert [infix to prefix](#).

In this article, we will discuss how to evaluate an expression written in prefix notation. The method is similar to evaluating a postfix expression. Please read [Evaluation of Postfix Expression](#) to know how to evaluate postfix expressions

Algorithm

EVALUATE_POSTFIX(String)

- Step 1: Put a pointer P at the end of the end
- Step 2: If character at P is an operand push it to Stack
- Step 3: If the character at P is an operator pop two elements from the Stack. Operate on these elements according to the operator, and push the result back to the Stack
- Step 4: Decrement P by 1 and go to Step 2 as long as there are characters left to be scanned in the expression.
- Step 5: The Result is stored at the top of the Stack, return it
- Step 6: End

Example to demonstrate working of the algorithm

Expression: +9*26

Character	Stack	Explanation
Scanned	(Front to Back)	

6	6	6 is an operand, push to Stack
2	6 2	2 is an operand, push to Stack
*	12 (6*2)	* is an operator, pop 6 and 2, multiply them and push result to Stack
9	12 9	9 is an operand, push to Stack
+	21 (12+9)	+ is an operator, pop 12 and 9 add them and push result to Stack

Result: 21

Examples:

Input : -+8/632

Output : 8

Input : -+7*45+2

Output : 25

Complexity The algorithm has linear complexity since we scan the expression once and perform at most $O(N)$ push and pop operations which take constant time.

Implementation of the algorithm is given below.

C++

```
// C++ program to evaluate a prefix expression.
#include <bits/stdc++.h>
using namespace std;

bool isOperand(char c)
{
    // If the character is a digit then it must
    // be an operand
    return isdigit(c);
}
```



```
}

double evaluatePrefix(string exprsn)
{
    stack<double> Stack;

    for (int j = exprsn.size() - 1; j >= 0; j--) {

        // Push operand to Stack
        // To convert exprsn[j] to digit subtract
        // '0' from exprsn[j].
        if (isOperand(exprsn[j]))
            Stack.push(exprsn[j] - '0');

        else {

            // Operator encountered
            // Pop two elements from Stack
            double o1 = Stack.top();
            Stack.pop();
            double o2 = Stack.top();
            Stack.pop();

            // Use switch case to operate on o1
            // and o2 and perform o1 0 o2.
            switch (exprsn[j]) {
                case '+':
                    Stack.push(o1 + o2);
                    break;
                case '-':
                    Stack.push(o1 - o2);
                    break;
                case '*':
                    Stack.push(o1 * o2);
                    break;
                case '/':
                    Stack.push(o1 / o2);
                    break;
            }
        }
    }

    return Stack.top();
}

// Driver code
int main()
{
```

```
    string exprsn = "+9*26";
    cout << evaluatePrefix(exprsn) << endl;
    return 0;
}
```

Java

```
// Java program to evaluate
// a prefix expression.
import java.io.*;
import java.util.*;

class GFG {

    static Boolean isOperand(char c)
    {
        // If the character is a digit
        // then it must be an operand
        if(c >= 48 && c <= 57)
            return true;
        else
            return false;
    }

    static double evaluatePrefix(String exprsn)
    {
        Stack<Double> Stack = new Stack<Double>();

        for (int j = exprsn.length() - 1; j >= 0; j--) {

            // Push operand to Stack
            // To convert exprsn[j] to digit subtract
            // '0' from exprsn[j].
            if (isOperand(exprsn.charAt(j)))
                Stack.push((double)(exprsn.charAt(j) - 48));

            else {

                // Operator encountered
                // Pop two elements from Stack
                double o1 = Stack.peek();
                Stack.pop();
                double o2 = Stack.peek();
                Stack.pop();

                // Use switch case to operate on o1
                // and o2 and perform o1 0 o2.
                switch (exprsn.charAt(j)) {
```

```
        case '+':
            Stack.push(o1 + o2);
            break;
        case '-':
            Stack.push(o1 - o2);
            break;
        case '*':
            Stack.push(o1 * o2);
            break;
        case '/':
            Stack.push(o1 / o2);
            break;
    }
}

return Stack.peek();
}

/* Driver program to test above function */
public static void main(String[] args)
{
    String exprsn = "+9/26";
    System.out.println(evaluatePrefix(exprsn));
}
}

// This code is contributed by Gitanjali
```

Output:

21

Note:

To perform more types of operations only the switch case table needs to be modified. This implementation works only for single digit operands. Multi-digit operands can be implemented if some character like space is used to separate the operands and operators.

Source

<https://www.geeksforgeeks.org/evaluation-prefix-expressions/>

Chapter 32

Expression Evaluation

Expression Evaluation - GeeksforGeeks

Evaluate an expression represented by a String. Expression can contain parentheses, you can assume parentheses are well-matched. For simplicity, you can assume only binary operations allowed are +, -, *, and /. Arithmetic Expressions can be written in one of three forms:

Infix Notation: Operators are written between the operands they operate on, e.g. $3 + 4$.

Prefix Notation: Operators are written before the operands, e.g. $+ 3 4$

Postfix Notation: Operators are written after operands.

Infix Expressions are harder for Computers to evaluate because of the additional work needed to decide precedence. Infix notation is how expressions are written and recognized by humans and, generally, input to programs. Given that they are harder to evaluate, they are generally converted to one of the two remaining forms. A very well known algorithm for converting an infix notation to a postfix notation is [Shunting Yard Algorithm by Edgar Dijkstra](#). This algorithm takes as input an Infix Expression and produces a queue that has this expression converted to a postfix notation. Same algorithm can be modified so that it outputs result of evaluation of expression instead of a queue. Trick is using two stacks instead of one, one for operands and one for operators. Algorithm was described succinctly on <http://www.cis.upenn.edu/~matuszek/cit594-2002/Assignments/5-expressions.htm>, and is re-produced here. (Note that credit for succinctness goes to author of said page)

1. While there are still tokens to be read in,
 - 1.1 Get the next token.
 - 1.2 If the token is:
 - 1.2.1 A number: push it onto the value stack.
 - 1.2.2 A variable: get its value, and push onto the value stack.
 - 1.2.3 A left parenthesis: push it onto the operator stack.
 - 1.2.4 A right parenthesis:
 - 1 While the thing on top of the operator stack is not a left parenthesis,

- 1 Pop the operator from the operator stack.
- 2 Pop the value stack twice, getting two operands.
- 3 Apply the operator to the operands, in the correct order.
- 4 Push the result onto the value stack.
- 2 Pop the left parenthesis from the operator stack, and discard it.
- 1.2.5 An operator (call it thisOp):
 - 1 While the operator stack is not empty, and the top thing on the operator stack has the same or greater precedence as thisOp,
 - 1 Pop the operator from the operator stack.
 - 2 Pop the value stack twice, getting two operands.
 - 3 Apply the operator to the operands, in the correct order.
 - 4 Push the result onto the value stack.
 - 2 Push thisOp onto the operator stack.
2. While the operator stack is not empty,
 - 1 Pop the operator from the operator stack.
 - 2 Pop the value stack twice, getting two operands.
 - 3 Apply the operator to the operands, in the correct order.
 - 4 Push the result onto the value stack.
3. At this point the operator stack should be empty, and the value stack should have only one value in it, which is the final result.

It should be clear that this algorithm runs in linear time – each number or operator is pushed onto and popped from Stack only once. Also see <http://www2.lawrence.edu/fast/GREGGJ/CMSC270/Infix.html>, <http://faculty.cs.niu.edu/~hutchins/csci241/eval.htm>.

Following is the implementation of above algorithm:

C++

```
// CPP program to evaluate a given
// expression where tokens are
// separated by space.
#include <bits/stdc++.h>
using namespace std;

// Function to find precedence of
// operators.
int precedence(char op){
    if(op == '+'||op == '-')
        return 1;
    if(op == '*'||op == '/')
        return 2;
    return 0;
}

// Function to perform arithmetic operations.
int applyOp(int a, int b, char op){
    switch(op){
```

```
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/': return a / b;
    }
}

// Function that returns value of
// expression after evaluation.
int evaluate(string tokens){
    int i;

    // stack to store integer values.
    stack <int> values;

    // stack to store operators.
    stack <char> ops;

    for(i = 0; i < tokens.length(); i++){

        // Current token is a whitespace,
        // skip it.
        if(tokens[i] == ' '){
            continue;
        }

        // Current token is an opening
        // brace, push it to 'ops'
        else if(tokens[i] == '('){
            ops.push(tokens[i]);
        }

        // Current token is a number, push
        // it to stack for numbers.
        else if(isdigit(tokens[i])){
            int val = 0;

            // There may be more than one
            // digits in number.
            while(i < tokens.length() &&
                  isdigit(tokens[i]))
            {
                val = (val*10) + (tokens[i]-'0');
                i++;
            }

            values.push(val);
        }
    }
}
```

```

// Closing brace encountered, solve
// entire brace.
else if(tokens[i] == ')')
{
    while(!ops.empty() && ops.top() != '(')
    {
        int val2 = values.top();
        values.pop();

        int val1 = values.top();
        values.pop();

        char op = ops.top();
        ops.pop();

        values.push(applyOp(val1, val2, op));
    }

    // pop opening brace.
    ops.pop();
}

// Current token is an operator.
else
{
    // While top of 'ops' has same or greater
    // precedence to current token, which
    // is an operator. Apply operator on top
    // of 'ops' to top two elements in values stack.
    while(!ops.empty() && precedence(ops.top())
        >= precedence(tokens[i])){
        int val2 = values.top();
        values.pop();

        int val1 = values.top();
        values.pop();

        char op = ops.top();
        ops.pop();

        values.push(applyOp(val1, val2, op));
    }

    // Push current token to 'ops'.
    ops.push(tokens[i]);
}
}

```

```
// Entire expression has been parsed at this
// point, apply remaining ops to remaining
// values.
while(!ops.empty()){
    int val2 = values.top();
    values.pop();

    int val1 = values.top();
    values.pop();

    char op = ops.top();
    ops.pop();

    values.push(applyOp(val1, val2, op));
}

// Top of 'values' contains result, return it.
return values.top();
}

int main() {
    cout << evaluate("10 + 2 * 6") << "\n";
    cout << evaluate("100 * 2 + 12") << "\n";
    cout << evaluate("100 * ( 2 + 12 )") << "\n";
    cout << evaluate("100 * ( 2 + 12 ) / 14");
    return 0;
}

// This code is contributed by Nikhil jindal.
```

Java

```
/* A Java program to evaluate a given expression where tokens are separated
by space.
Test Cases:
    "10 + 2 * 6"          ---> 22
    "100 * 2 + 12"        ---> 212
    "100 * ( 2 + 12 )"    ---> 1400
    "100 * ( 2 + 12 ) / 14" ---> 100
*/
import java.util.Stack;

public class EvaluateString
{
    public static int evaluate(String expression)
    {
        char[] tokens = expression.toCharArray();
```



```
// Stack for numbers: 'values'
Stack<Integer> values = new Stack<Integer>();

// Stack for Operators: 'ops'
Stack<Character> ops = new Stack<Character>();

for (int i = 0; i < tokens.length; i++)
{
    // Current token is a whitespace, skip it
    if (tokens[i] == ' ')
        continue;

    // Current token is a number, push it to stack for numbers
    if (tokens[i] >= '0' && tokens[i] <= '9')
    {
        StringBuffer sbuf = new StringBuffer();
        // There may be more than one digits in number
        while (i < tokens.length && tokens[i] >= '0' && tokens[i] <= '9')
            sbuf.append(tokens[i++]);
        values.push(Integer.parseInt(sbuf.toString()));
    }

    // Current token is an opening brace, push it to 'ops'
    else if (tokens[i] == '(')
        ops.push(tokens[i]);

    // Closing brace encountered, solve entire brace
    else if (tokens[i] == ')')
    {
        while (ops.peek() != '(')
            values.push(applyOp(ops.pop(), values.pop(), values.pop()));
        ops.pop();
    }

    // Current token is an operator.
    else if (tokens[i] == '+' || tokens[i] == '-' ||
             tokens[i] == '*' || tokens[i] == '/')
    {
        // While top of 'ops' has same or greater precedence to current
        // token, which is an operator. Apply operator on top of 'ops'
        // to top two elements in values stack
        while (!ops.empty() && hasPrecedence(tokens[i], ops.peek()))
            values.push(applyOp(ops.pop(), values.pop(), values.pop()));

        // Push current token to 'ops'.
        ops.push(tokens[i]);
    }
}
```

```
// Entire expression has been parsed at this point, apply remaining
// ops to remaining values
while (!ops.empty())
    values.push(applyOp(ops.pop(), values.pop(), values.pop()));

// Top of 'values' contains result, return it
return values.pop();
}

// Returns true if 'op2' has higher or same precedence as 'op1',
// otherwise returns false.
public static boolean hasPrecedence(char op1, char op2)
{
    if (op2 == '(' || op2 == ')')
        return false;
    if ((op1 == '*' || op1 == '/') && (op2 == '+' || op2 == '-'))
        return false;
    else
        return true;
}

// A utility method to apply an operator 'op' on operands 'a'
// and 'b'. Return the result.
public static int applyOp(char op, int b, int a)
{
    switch (op)
    {
        case '+':
            return a + b;
        case '-':
            return a - b;
        case '*':
            return a * b;
        case '/':
            if (b == 0)
                throw new
                    UnsupportedOperationException("Cannot divide by zero");
            return a / b;
    }
    return 0;
}

// Driver method to test above methods
public static void main(String[] args)
{
    System.out.println(EvaluateString.evaluate("10 + 2 * 6"));
    System.out.println(EvaluateString.evaluate("100 * 2 + 12"));
}
```

```
        System.out.println(EvaluateString.evaluate("100 * ( 2 + 12 )"));
        System.out.println(EvaluateString.evaluate("100 * ( 2 + 12 ) / 14"));
    }
}
```

Output:

```
22
212
1400
100
```

Time Complexity: $O(n)$

Space Complexity: $O(n)$

See [this](#) for a sample run with more test cases.

This article is compiled by **Ciphe**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : [nik1996](#)

Source

<https://www.geeksforgeeks.org/expression-evaluation/>

Chapter 33

Expression contains redundant bracket or not

Expression contains redundant bracket or not - GeeksforGeeks

Given a string of balanced expression, find if it contains a redundant parenthesis or not. A set of parenthesis are redundant if same sub-expression is surrounded by unnecessary or multiple brackets. Print 'Yes' if redundant else 'No'.

Note: Expression may contain '+', '*', '-', and '/' operators. Given expression is **valid** and there are **no white** spaces present.

Input:

((a+b))

(a+(b)/c)

(a+b*(c-d))

Output:

Yes

Yes

No

Explanation:

1. ((a+b)) can reduced to (a+b), this Redundant
2. (a+(b)/c) can reduced to (a+b/c) because b is surrounded by () which is redundant.
3. (a+b*(c-d)) doesn't have any redundant or multiple brackets.

The idea is to use stack which is discussed in [this](#) article. For any sub-expression of expression, if we able to pick any sub-expression of expression surrounded by (), then we again left with () as part of string, we have redundant braces.

We iterate through the given expression and for each character in the expression, if the character is a open parenthesis '(' or any of the operators or operands, we push it to the stack. If the character is close parenthesis ')', then pop characters from the stack till matching open parenthesis '(' is found.

Now for redundancy two condition will arise while popping-

1. If immediate pop hits a open parenthesis '(', then we have found a duplicate parenthesis. For example, $((a+b))+c$ has duplicate brackets around $a+b$. When we reach second "(" after $a+b$, we have "(" in the stack. Since the top of stack is a opening bracket, we conclude that there are duplicate brackets.
2. If immediate pop doesn't hit any operand('(', '+', '/', '-') then it indicates the presence of unwanted brackets surrounded by expression. For instance, $(a)+b$ contain unwanted () around a thus it is redundant.

```
/* C++ Program to check whether valid
expression is redundant or not*/
#include <iostream>
#include <stack>
using namespace std;

// Function to check redundant brackets in a
// balanced expression
bool checkRedundancy(string& str)
{
    // create a stack of characters
    stack<char> st;

    // Iterate through the given expression
    for (auto& ch : str) {

        // if current character is close parenthesis ')'
        if (ch == ')') {
            char top = st.top();
            st.pop();

            // If immediate pop have open parenthesis '('
            // duplicate brackets found
            bool flag = true;

            while (top != '(') {

                // Check for operators in expression
                if (top == '+' || top == '-' ||
                    top == '*' || top == '/')
                    flag = false;

                // Fetch top element of stack
                top = st.top();
            }
        }
    }
}
```

```
        st.pop();
    }

    // If operators not found
    if (flag == true)
        return true;
}

else
    st.push(ch); // push open parenthesis '(',
                // operators and operands to stack
}
return false;
}

// Function to check redundant brackets
void findRedundant(string& str)
{
    bool ans = checkRedundancy(str);
    if (ans == true)
        cout << "Yes\n";
    else
        cout << "No\n";
}

// Driver code
int main()
{
    string str = "((a+b))";
    findRedundant(str);

    str = "(a+(b)/c)";
    findRedundant(str);

    str = "(a+b*(c-d))";
    findRedundant(str);

    return 0;
}
```

Output

Yes

Yes

No

Time complexity: $O(n)$

Auxiliary space: $O(n)$

Source

<https://www.geeksforgeeks.org/expression-contains-redundant-bracket-not/>

Chapter 34

Find if an expression has duplicate parenthesis or not

Find if an expression has duplicate parenthesis or not - GeeksforGeeks

Given an balanced expression, find if it contains duplicate parenthesis or not. A set of parenthesis are duplicate if same sub expression is surrounded by multiple parenthesis.

Examples:

Below expressions have duplicate parenthesis -
`((a+b)+((c+d)))`

The subexpression "c+d" is surrounded by two pairs of brackets.

`((a+(b)))+(c+d))`

The subexpression "a+(b)" is surrounded by two pairs of brackets.

`((a+(b))+c+d))`

The whole expression is surrounded by two pairs of brackets.

Below expressions don't have any duplicate parenthesis -
`((a+b)+(c+d))`

No subexpression is surrounded by duplicate brackets.

`((a+(b)))+(c+d))`

No subexpression is surrounded by duplicate brackets.

It may be assumed that the given expression is valid and there are not any white spaces present.

The idea is to use stack. Iterate through the given expression and for each character in the expression, if the character is a open parenthesis '(' or any of the operators or operands, push it to the top of the stack. If the character is close parenthesis ')', then pop characters from the stack till matching open parenthesis '(' is found and a counter is used, whose value is incremented for every character encountered till the opening parenthesis '(' is found. If the number of characters encountered between the opening and closing parenthesis pair, which is equal to the value of the counter, is less than or equal to 1, then a pair of duplicate parenthesis is found else there is no occurrence of redundant parenthesis pairs. For example, (((a+b))+c) has duplicate brackets around "a+b". When the second ")" after a+b is encountered, the stack contains "((" . Since the top of stack is a opening bracket, it can be concluded that there are duplicate brackets.

Below is C++ implementation of above idea :

```
// C++ program to find duplicate parenthesis in a
// balanced expression
#include <iostream>
#include <stack>
using namespace std;

// Function to find duplicate parenthesis in a
// balanced expression
bool findDuplicateparenthesis(string str)
{
    // create a stack of characters
    stack<char> Stack;

    // Iterate through the given expression
    for (char ch : str)
    {
        // if current character is close parenthesis ')'
        if (ch == ')')
        {
            // pop character from the stack
            char top = Stack.top();
            Stack.pop();

            // stores the number of characters between a
            // closing and opening parenthesis
            // if this count is less than or equal to 1
            // then the brackets are redundant else not
            int elementsInside = 0;
            while (top != '(')
            {
                elementsInside++;
                top = Stack.top();
            }
        }
    }
}
```

```
        Stack.pop();
    }
    if(elementsInside <= 1) {
        return 1;
    }
}

// push open parenthesis '(', operators and
// operands to stack
else
    Stack.push(ch);
}

// No duplicates found
return false;
}

// Driver code
int main()
{
    // input balanced expression
    string str = "((a+(b))+(c+d))";

    if (findDuplicateparenthesis(str))
        cout << "Duplicate Found ";
    else
        cout << "No Duplicates Found ";

    return 0;
}
```

Output:

Duplicate Found

Time complexity of above solution is $O(n)$.
Auxiliary space used by the program is $O(n)$.

Improved By : [sirjan13](#)

Source

<https://www.geeksforgeeks.org/find-expression-duplicate-parenthesis-not/>

Chapter 35

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

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

Given a string with brackets. If the start index of the open bracket is given, find the index of the closing bracket.

Examples:

```
Input : string = [ABC[23]][89]
        index = 0
```

```
Output : 8
```

The opening bracket at index 0 corresponds to closing bracket at index 8.

The idea is to use [Stack data structure](#). We traverse given expression from given index and keep pushing starting brackets. Whenever we encounter a closing bracket, we pop a starting bracket. If stack becomes empty at any moment, we return that index.

C++

```
// CPP program to find index of closing
// bracket for given opening bracket.
#include <bits/stdc++.h>
using namespace std;

// Function to find index of closing
// bracket for given opening bracket.
void test(string expression, int index){
```

```
int i;

// If index given is invalid and is
// not an opening bracket.
if(expression[index]!='['){
    cout << expression << ", " <<
        index << ": -1\n";
    return;
}

// Stack to store opening brackets.
stack <int> st;

// Traverse through string starting from
// given index.
for(i = index; i < expression.length(); i++){

    // If current character is an
    // opening bracket push it in stack.
    if(expression[i] == '[')
        st.push(expression[i]);

    // If current character is a closing
    // bracket, pop from stack. If stack
    // is empty, then this closing
    // bracket is required bracket.
    else if(expression[i] == ' '){
        st.pop();
        if(st.empty()){
            cout << expression << ", " <<
                index << ": " << i << "\n";
            return;
        }
    }
}

// If no matching closing bracket
// is found.
cout << expression << ", " <<
    index << ": -1\n";
}

// Driver Code
int main() {
    test("[ABC[23]][89]", 0); // should be 8
    test("[ABC[23]][89]", 4); // should be 7
    test("[ABC[23]][89]", 9); // should be 12
    test("[ABC[23]][89]", 1); // No matching bracket
```

```
    return 0;
}

// This code is contributed by Nikhil Jindal.
```

Python

```
# Python program to find index of closing
# bracket for a given opening bracket.
from collections import deque

def getIndex(s, i):

    # If input is invalid.
    if s[i] != '[':
        return -1

    # Create a deque to use it as a stack.
    d = deque()

    # Traverse through all elements
    # starting from i.
    for k in range(i, len(s)):

        # Pop a starting bracket
        # for every closing bracket
        if s[k] == ']':
            d.popleft()

        # Push all starting brackets
        elif s[k] == '[':
            d.append(s[i])

        # If deque becomes empty
        if not d:
            return k

    return -1

# Driver code to test above method.
def test(s, i):
    matching_index = getIndex(s, i)
    print(s + ", " + str(i) + ": " + str(matching_index))

def main():
    test("[ABC[23]][89]", 0) # should be 8
    test("[ABC[23]][89]", 4) # should be 7
    test("[ABC[23]][89]", 9) # should be 12
```

```
test("ABC[23]][89]", 1) # No matching bracket

if __name__ == "__main__":
    main()
```

```
Output : [ABC[23]][89], 0: 8
Output : [ABC[23]][89], 4: 7
Output : [ABC[23]][89], 9: 12
Output : [ABC[23]][89], 1: -1
```

Time Complexity: $O(n)$

Auxiliary Space: $O(n)$

Improved By : [nik1996](#)

Source

<https://www.geeksforgeeks.org/find-index-closing-bracket-given-opening-bracket-expression/>

Chapter 36

Find maximum depth of nested parenthesis in a string

Find maximum depth of nested parenthesis in a string - GeeksforGeeks

We are given a string having parenthesis like below

“(((X)) (((Y))))”

We need to find the maximum depth of balanced parenthesis, like 4 in above example. Since ‘Y’ is surrounded by 4 balanced parenthesis.

If parenthesis are unbalanced then return -1.

Examples :

Input : S = "(a(b) (c) (d(e(f)g)h) I (j(k)l)m)";
Output : 4

Input : S = "(p((q)) ((s)t))";
Output : 3

Input : S = "";
Output : 0

Input : S = "b) (c) ()";
Output : -1

Input : S = "(b) ((c) ()"
Output : -1

Method 1 (Uses Stack)

A simple solution is to use a stack that keeps track of current open brackets.

- 1) Create a stack.
- 2) Traverse the string, do following for every character
 - a) If current character is '(' push it to the stack .
 - b) If character is ')', pop an element.
 - c) Maintain maximum count during the traversal.

Time Complexity : $O(n)$

Auxiliary Space : $O(n)$

Method 2 ($O(1)$ auxiliary space)

This can also be done without using stack.

- 1) Take two variables max and current_max, initialize both of them as 0.
- 2) Traverse the string, do following for every character
 - a) If current character is '(', increment current_max and update max value if required.
 - b) If character is ')'. Check if current_max is positive or not (this condition ensure that parenthesis are balanced). If positive that means we previously had a '(' character so decrement current_max without worry. If not positive then the parenthesis are not balanced. Thus return -1.
- 3) If current_max is not 0, then return -1 to ensure that the parenthesis are balanced. Else return max

Below is the implementation of above algorithm.

C/C++

```
// A C++ program to find the maximum depth of nested
// parenthesis in a given expression
#include <iostream>
using namespace std;

// function takes a string and returns the
// maximum depth nested parenthesis
int maxDepth(string S)
{
    int current_max = 0; // current count
    int max = 0;        // overall maximum count
    int n = S.length();

    // Traverse the input string
    for (int i = 0; i < n; i++)
    {
        if (S[i] == '(')
```



```
{
    current_max++;

    // update max if required
    if (current_max > max)
        max = current_max;
}
else if (S[i] == ')')
{
    if (current_max > 0)
        current_max--;
    else
        return -1;
}
}

// finally check for unbalanced string
if (current_max != 0)
    return -1;

return max;
}

// Driver program
int main()
{
    string s = "( ((X)) (((Y))) )";
    cout << maxDepth(s);
    return 0;
}
```

Python

```
# A Python program to find the maximum depth of nested
# parenthesis in a given expression

# function takes a string and returns the
# maximum depth nested parenthesis
def maxDepth(S):
    current_max = 0
    max = 0
    n = len(S)

    # Traverse the input string
    for i in xrange(n):
        if S[i] == '(':
            current_max += 1
```

```
        if current_max > max:
            max = current_max
    elif S[i] == ')':
        if current_max > 0:
            current_max -= 1
        else:
            return -1

    # finally check for unbalanced string
    if current_max != 0:
        return -1

    return max

# Driver program
s = "( (X) ((Y))) )"
print maxDepth(s)

# This code is contributed by BHAVYA JAIN
```

PHP

```
<?php
// A PHP program to find the
// maximum depth of nested
// parenthesis in a given
// expression

// function takes a string
// and returns the maximum
// depth nested parenthesis
function maxDepth($S)
{
    // current count
    $current_max = 0;

    // overall maximum count
    $max = 0;
    $n = strlen($S);

    // Traverse the input string
    for ($i = 0; $i < $n; $i++)
    {
        if ($S[$i] == '(')
        {
            $current_max++;

            // update max if required
```

```
        if ($current_max> $max)
            $max = $current_max;
    }

    else if ($S[$i] == ')')
    {
        if ($current_max>0)
            $current_max--;
        else
            return -1;
    }
}

// finally check for
// unbalanced string
if ($current_max != 0)
    return -1;

return $max;
}

// Driver Code
$s = "( (X)) ((Y))) ";
echo maxDepth($s);

// This code is contributed by mits
?>
```

Output :

4

Time Complexity : $O(n)$

Auxiliary Space : $O(1)$

This article is contributed by **Gaurav Sharma**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : [Mithun Kumar](#)

Source

<https://www.geeksforgeeks.org/find-maximum-depth-nested-parenthesis-string/>

Chapter 37

Find maximum difference between nearest left and right smaller elements

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

Given an array of integers, the task is to find the maximum absolute difference between the nearest left and the right smaller element of every element in the array.

Note: If there is no smaller element on right side or left side of any element then we take zero as the smaller element. For example for the leftmost element, the nearest smaller element on the left side is considered as 0. Similarly, for rightmost elements, the smaller element on the right side is considered as 0.

Examples:

```
Input : arr[] = {2, 1, 8}
Output : 1
Left smaller  LS[] {0, 0, 1}
Right smaller RS[] {1, 0, 0}
Maximum Diff of abs(LS[i] - RS[i]) = 1
```

```
Input  : arr[] = {2, 4, 8, 7, 7, 9, 3}
Output : 4
Left smaller  LS[] = {0, 2, 4, 4, 4, 7, 2}
Right smaller RS[] = {0, 3, 7, 3, 3, 3, 0}
Maximum Diff of abs(LS[i] - RS[i]) = 7 - 3 = 4
```

```
Input : arr[] = {5, 1, 9, 2, 5, 1, 7}
Output : 1
```

A **simple solution** is to find the nearest left and right smaller elements for every element and then update the maximum difference between left and right smaller element, this takes $O(n^2)$ time.

An **efficient solution** takes $O(n)$ time. We use a [stack](#). The idea is based on the approach discussed in [next greater element article](#). The interesting part here is we compute both left smaller and right smaller using same function.

Let input array be 'arr[]' and size of array be 'n'

Find all smaller element on left side

1. Create a new empty stack S and an array LS[]
2. For every element 'arr[i]' in the input arr[],
where 'i' goes from 0 to n-1.
 - a) while S is nonempty and the top element of S is greater than or equal to 'arr[i]':
pop S
 - b) if S is empty:
'arr[i]' has no preceding smaller value
LS[i] = 0
 - c) else:
the nearest smaller value to 'arr[i]' is top of stack
LS[i] = s.top()
 - d) push 'arr[i]' onto S

Find all smaller element on right side

3. First reverse array arr[]. After reversing the array, right smaller become left smaller.
4. Create an array RRS[] and repeat steps 1 and 2 to fill RRS (in-place of LS).
5. Initialize result as -1 and do following for every element arr[i]. In the reversed array right smaller for arr[i] is stored at RRS[n-i-1]
return result = max(result, LS[i]-RRS[n-i-1])

Below is implementation of above idea

C/C++

```
// C++ program to find the difference b/w left and  
// right smaller element of every element in array
```

```
#include<bits/stdc++.h>
using namespace std;

// Function to fill left smaller element for every
// element of arr[0..n-1]. These values are filled
// in SE[0..n-1]
void leftSmaller(int arr[], int n, int SE[])
{
    // Create an empty stack
    stack<int>S;

    // Traverse all array elements
    // compute nearest smaller elements of every element
    for (int i=0; i<n; i++)
    {
        // Keep removing top element from S while the top
        // element is greater than or equal to arr[i]
        while (!S.empty() && S.top() >= arr[i])
            S.pop();

        // Store the smaller element of current element
        if (!S.empty())
            SE[i] = S.top();

        // If all elements in S were greater than arr[i]
        else
            SE[i] = 0;

        // Push this element
        S.push(arr[i]);
    }
}

// Function returns maximum difference b/w Left &
// right smaller element
int findMaxDiff(int arr[], int n)
{
    int LS[n]; // To store left smaller elements

    // find left smaller element of every element
    leftSmaller(arr, n, LS);

    // find right smaller element of every element
    // first reverse the array and do the same process
    int RRS[n]; // To store right smaller elements in
                // reverse array
    reverse(arr, arr + n);
    leftSmaller(arr, n, RRS);
```

```
// find maximum absolute difference b/w LS & RRS
// In the reversed array right smaller for arr[i] is
// stored at RRS[n-i-1]
int result = -1;
for (int i=0 ; i< n ; i++)
    result = max(result, abs(LS[i] - RRS[n-1-i]));

// return maximum difference b/w LS & RRS
return result;
}

// Driver program
int main()
{
    int arr[] = {2, 4, 8, 7, 7, 9, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Maximum diff : "
        << findMaxDiff(arr, n) << endl;
    return 0;
}
```

Python

```
# Python program to find the difference b/w left and
# right smaller element of every element in the array

# Function to fill left smaller element for every
# element of arr[0..n-1]. These values are filled
# in SE[0..n-1]
def leftsmaller(arr, n, SE):

    # create an empty stack
    sta = []
    # Traverse all array elements
    # compute nearest smaller elements of every element
    for i in range(n):

        # Keep removing top element from S while the top
        # element is greater than or equal to arr[i]
        while(sta != [] and sta[len(sta)-1] >= arr[i]):
            sta.pop()

        # Store the smaller element of current element
        if(sta != []):
            SE[i]=sta[len(sta)-1]
        # If all elements in S were greater than arr[i]
        else:
```

```
SE[i]=0

# push this element
sta.append(arr[i])

# Function returns maximum difference b/w Left &
# right smaller elemen
def findMaxDiff(arr, n):
    ls=[0]*n # to store left smaller elements
    rs=[0]*n # to store right smaller elements

    # find left smaller elements of every element
    leftsmaller(arr, n, ls)

    # find right smaller element of every element
    # by sending reverse of array
    leftsmaller(arr[::-1], n, rs)

    # find maximum absolute difference b/w LS & RRS
    # In the reversed array right smaller for arr[i] is
    # stored at RRS[n-i-1]
    res = -1
    for i in range(n):
        res = max(res, abs(ls[i] - rs[n-1-i]))
    # return maximum difference b/w LS & RRS
    return res

# Driver Program
if __name__=='__main__':
    arr = [2, 4, 8, 7, 7, 9, 3]
    print "Maximum Diff :", findMaxDiff(arr, len(arr))

#Contributed By: Harshit Sidhwa
```

PHP

```
<?php
// Function to fill left smaller
// element for every element of
// arr[0..n-1]. These values are
// filled in SE[0..n-1]
function leftSmaller(&$arr, $n, &$SE)
{
    $S = array();

    // Traverse all array elements
    // compute nearest smaller
```



```
// elements of every element
for ($i = 0; $i < $n; $i++)
{
    // Keep removing top element
    // from S while the top element
    // is greater than or equal to arr[i]
    while (!empty($S) && max($S) >= $arr[$i])
        array_pop($S);

    // Store the smaller element
    // of current element
    if (!empty($S))
        $SE[$i] = max($S);

    // If all elements in S were
    // greater than arr[i]
    else
        $SE[$i] = 0;

    // Push this element
    array_push($S, $arr[$i]);
}
}
```

```
// Function returns maximum
// difference b/w Left &
// right smaller element
function findMaxDiff(&$arr, $n)
{
    // To store left smaller elements
    $LS = array_fill(0, $n, NULL);

    // find left smaller element
    // of every element
    leftSmaller($arr, $n, $LS);

    // find right smaller element
    // of every element first reverse
    // the array and do the same process

    // To store right smaller
    // elements in reverse array
    $RRS = array_fill(0, $n, NULL);

    $k = 0;
    for($i = $n - 1; $i >= 0; $i--)
        $x[$k++] = $arr[$i];
    leftSmaller($x, $n, $RRS);
}
```

```
// find maximum absolute difference
// b/w LS & RRS. In the reversed
// array right smaller for arr[i]
// is stored at RRS[n-i-1]
$result = -1;
for ($i = 0 ; $i < $n ; $i++)
    $result = max($result, abs($LS[$i] -
        $RRS[$n - 1 - $i]));

// return maximum difference
// b/w LS & RRS
return $result;
}

// Driver Code
$arr = array(2, 4, 8, 7, 7, 9, 3);
$n = sizeof($arr);
echo "Maximum diff : " .
    findMaxDiff($arr, $n) . "\n";

// This code is contributed
// by ChitraNayal
?>
```

Output:

Maximum Diff : 4

Time complexity: $O(n)$

Improved By : [ChitraNayal](#)

Source

<https://www.geeksforgeeks.org/find-maximum-difference-between-nearest-left-and-right-smaller-elements/>

Chapter 38

Find maximum of minimum for every window size in a given array

Find maximum of minimum for every window size in a given array - GeeksforGeeks

Given an integer array of size n, find the maximum of the minimum's of every window size in the array. Note that window size varies from 1 to n.

Example:

Input: arr[] = {10, 20, 30, 50, 10, 70, 30}
Output: 70, 30, 20, 10, 10, 10, 10

First element in output indicates maximum of minimums of all windows of size 1.

Minimums of windows of size 1 are {10}, {20}, {30}, {50}, {10}, {70} and {30}. Maximum of these minimums is 70

Second element in output indicates maximum of minimums of all windows of size 2.

Minimums of windows of size 2 are {10}, {20}, {30}, {10}, {10}, and {30}. Maximum of these minimums is 30

Third element in output indicates maximum of minimums of all windows of size 3.

Minimums of windows of size 3 are {10}, {20}, {10}, {10} and {10}. Maximum of these minimums is 20

Similarly other elements of output are computed.

A **Simple Solution** is to go through all windows of every size, find maximum of all windows. Below is implementation of this idea.

C++

```
// A naive method to find maximum of minimum of all windows of
// different sizes
#include <iostream>
#include <climits>
using namespace std;

void printMaxOfMin(int arr[], int n)
{
    // Consider all windows of different sizes starting
    // from size 1
    for (int k=1; k<=n; k++)
    {
        // Initialize max of min for current window size k
        int maxOfMin = INT_MIN;

        // Traverse through all windows of current size k
        for (int i = 0; i <= n-k; i++)
        {
            // Find minimum of current window
            int min = arr[i];
            for (int j = 1; j < k; j++)
            {
                if (arr[i+j] < min)
                    min = arr[i+j];
            }

            // Update maxOfMin if required
            if (min > maxOfMin)
                maxOfMin = min;
        }

        // Print max of min for current window size
        cout << maxOfMin << " ";
    }
}

// Driver program
int main()
{
    int arr[] = {10, 20, 30, 50, 10, 70, 30};
    int n = sizeof(arr)/sizeof(arr[0]);
    printMaxOfMin(arr, n);
    return 0;
}
```

```
}
```

Java

```
// A naive method to find maximum of minimum of all windows of  
// different sizes
```

```
class Test  
{  
    static int arr[] = {10, 20, 30, 50, 10, 70, 30};  
  
    static void printMaxOfMin(int n)  
    {  
        // Consider all windows of different sizes starting  
        // from size 1  
        for (int k=1; k<=n; k++)  
        {  
            // Initialize max of min for current window size k  
            int maxOfMin = Integer.MIN_VALUE;  
  
            // Traverse through all windows of current size k  
            for (int i = 0; i <= n-k; i++)  
            {  
                // Find minimum of current window  
                int min = arr[i];  
                for (int j = 1; j < k; j++)  
                {  
                    if (arr[i+j] < min)  
                        min = arr[i+j];  
                }  
  
                // Update maxOfMin if required  
                if (min > maxOfMin)  
                    maxOfMin = min;  
            }  
  
            // Print max of min for current window size  
            System.out.print(maxOfMin + " ");  
        }  
    }  
  
    // Driver method  
    public static void main(String[] args)  
    {  
        printMaxOfMin(arr.length);  
    }  
}
```

C#

```
// C# program using Naive approach to find
// maximum of minimum of all windows of
// different sizes
using System;

class GFG{

    static int []arr = {10, 20, 30, 50, 10, 70, 30};

    // Function to print maximum of minimum
    static void printMaxOfMin(int n)
    {

        // Consider all windows of different
        // sizes starting from size 1
        for (int k = 1; k <= n; k++)
        {

            // Initialize max of min for
            // current window size k
            int maxOfMin = int.MinValue;

            // Traverse through all windows
            // of current size k
            for (int i = 0; i <= n - k; i++)
            {

                // Find minimum of current window
                int min = arr[i];
                for (int j = 1; j < k; j++)
                {
                    if (arr[i + j] < min)
                        min = arr[i + j];
                }

                // Update maxOfMin if required
                if (min > maxOfMin)
                    maxOfMin = min;
            }

            // Print max of min for current window size
            Console.Write(maxOfMin + " ");
        }
    }

    // Driver Code
```

```
public static void Main()
{
    printMaxOfMin(arr.Length);
}

// This code is contributed by Sam007.
```

PHP

```
<?php
// PHP program to find maximum of
// minimum of all windows of
// different sizes

// Method to find maximum of
// minimum of all windows of
// different sizes
function printMaxOfMin($arr, $n)
{
    // Consider all windows of
    // different sizes starting
    // from size 1
    for($k = 1; $k <= $n; $k++)
    {
        // Initialize max of min for
        // current window size k
        $maxOfMin = PHP_INT_MIN;

        // Traverse through all windows
        // of current size k
        for ($i = 0; $i <= $n-$k; $i++)
        {
            // Find minimum of current window
            $min = $arr[$i];
            for ($j = 1; $j < $k; $j++)
            {
                if ($arr[$i + $j] < $min)
                    $min = $arr[$i + $j];
            }

            // Update maxOfMin
            // if required
            if ($min > $maxOfMin)
                $maxOfMin = $min;
        }
    }
}
```

```

    }

    // Print max of min for
    // current window size
    echo $maxOfMin , " ";
}

// Driver Code
$arr= array(10, 20, 30, 50, 10, 70, 30);
$n = sizeof($arr);
printMaxOfMin($arr, $n);

// This code is contributed by nitin mittal.
?>

```

Output:

```
70 30 20 10 10 10 10
```

Time complexity of above solution can be upper bounded by $O(n^3)$.

We can solve this problem in $O(n)$ time using an **Efficient Solution**. The idea is to extra space. Below are detailed steps.

Step 1: Find indexes of next smaller and previous smaller for every element. Next smaller is the nearest smallest element on right side of $arr[i]$. Similarly, previous smaller element is the nearest smallest element on left side of $arr[i]$.

If there is no smaller element on right side, then next smaller is n . If there is no smaller on left side, then previous smaller is -1 .

For input $\{10, 20, 30, 50, 10, 70, 30\}$, array of indexes of next smaller is $\{7, 4, 4, 4, 7, 6, 7\}$. For input $\{10, 20, 30, 50, 10, 70, 30\}$, array of indexes of previous smaller is $\{-1, 0, 1, 2, -1, 4, 4\}$

This step can be done in $O(n)$ time using the approach discussed in [next greater element](#).

Step 2: Once we have indexes of next and previous smaller, we know that $arr[i]$ is a minimum of a window of length " $right[i] - left[i] - 1$ ". Lengths of windows for which the elements are minimum are $\{7, 3, 2, 1, 7, 1, 2\}$. This array indicates, first element is minimum in window of size 7, second element is minimum in window of size 3, and so on.

Create an auxiliary array $ans[n+1]$ to store the result. Values in $ans[]$ can be filled by iterating through $right[]$ and $left[]$

```

for (int i=0; i < n; i++)
{
    // length of the interval
    int len = right[i] - left[i] - 1;

```



```
// a[i] is the possible answer for
// this length len interval
ans[len] = max(ans[len], arr[i]);
}
```

We get the ans[] array as {0, 70, 30, 20, 0, 0, 0, 10}. Note that ans[0] or answer for length 0 is useless.

Step 3: Some entries in ans[] are 0 and yet to be filled. For example, we know maximum of minimum for lengths 1, 2, 3 and 7 are 70, 30, 20 and 10 respectively, but we don't know the same for lengths 4, 5 and 6.

Below are few important observations to fill remaining entries

a) Result for length i, i.e. ans[i] would always be greater or same as result for length i+1, i.e., ans[i+1].

b) If ans[i] is not filled it means there is no direct element which is minimum of length i and therefore either the element of length ans[i+1], or ans[i+2], and so on is same as ans[i]

So we fill rest of the entries using below loop.

```
for (int i=n-1; i>=1; i--)
    ans[i] = max(ans[i], ans[i+1]);
```

Below is implementation of above algorithm.

C++

```
// An efficient C++ program to find maximum of all minimums of
// windows of different sizes
#include <iostream>
#include<stack>
using namespace std;

void printMaxOfMin(int arr[], int n)
{
    stack<int> s; // Used to find previous and next smaller

    // Arrays to store previous and next smaller
    int left[n+1];
    int right[n+1];

    // Initialize elements of left[] and right[]
    for (int i=0; i<n; i++)
    {
        left[i] = -1;
        right[i] = n;
    }
}
```

```
// Fill elements of left[] using logic discussed on
// https://www.geeksforgeeks.org/next-greater-element/
for (int i=0; i<n; i++)
{
    while (!s.empty() && arr[s.top()] >= arr[i])
        s.pop();

    if (!s.empty())
        left[i] = s.top();

    s.push(i);
}

// Empty the stack as stack is going to be used for right[]
while (!s.empty())
    s.pop();

// Fill elements of right[] using same logic
for (int i = n-1 ; i>=0 ; i-- )
{
    while (!s.empty() && arr[s.top()] >= arr[i])
        s.pop();

    if(!s.empty())
        right[i] = s.top();

    s.push(i);
}

// Create and initialize answer array
int ans[n+1];
for (int i=0; i<=n; i++)
    ans[i] = 0;

// Fill answer array by comparing minimums of all
// lengths computed using left[] and right[]
for (int i=0; i<n; i++)
{
    // length of the interval
    int len = right[i] - left[i] - 1;

    // arr[i] is a possible answer for this length
    // 'len' interval, check if arr[i] is more than
    // max for 'len'
    ans[len] = max(ans[len], arr[i]);
}

// Some entries in ans[] may not be filled yet. Fill
```

```
// them by taking values from right side of ans[]
for (int i=n-1; i>=1; i--)
    ans[i] = max(ans[i], ans[i+1]);

// Print the result
for (int i=1; i<=n; i++)
    cout << ans[i] << " ";
}

// Driver program
int main()
{
    int arr[] = {10, 20, 30, 50, 10, 70, 30};
    int n = sizeof(arr)/sizeof(arr[0]);
    printMaxOfMin(arr, n);
    return 0;
}
```

Java

```
// An efficient Java program to find maximum of all minimums of
// windows of different size

import java.util.Stack;

class Test
{
    static int arr[] = {10, 20, 30, 50, 10, 70, 30};

    static void printMaxOfMin(int n)
    {
        // Used to find previous and next smaller
        Stack<Integer> s = new Stack<>();

        // Arrays to store previous and next smaller
        int left[] = new int[n+1];
        int right[] = new int[n+1];

        // Initialize elements of left[] and right[]
        for (int i=0; i<n; i++)
        {
            left[i] = -1;
            right[i] = n;
        }

        // Fill elements of left[] using logic discussed on
        // https://www.geeksforgeeks.org/next-greater-element/
        for (int i=0; i<n; i++)
```

```
{
    while (!s.empty() && arr[s.peek()] >= arr[i])
        s.pop();

    if (!s.empty())
        left[i] = s.peek();

    s.push(i);
}

// Empty the stack as stack is going to be used for right[]
while (!s.empty())
    s.pop();

// Fill elements of right[] using same logic
for (int i = n-1 ; i>=0 ; i-- )
{
    while (!s.empty() && arr[s.peek()] >= arr[i])
        s.pop();

    if(!s.empty())
        right[i] = s.peek();

    s.push(i);
}

// Create and initialize answer array
int ans[] = new int[n+1];
for (int i=0; i<=n; i++)
    ans[i] = 0;

// Fill answer array by comparing minimums of all
// lengths computed using left[] and right[]
for (int i=0; i<n; i++)
{
    // length of the interval
    int len = right[i] - left[i] - 1;

    // arr[i] is a possible answer for this length
    // 'len' interval, check if arr[i] is more than
    // max for 'len'
    ans[len] = Math.max(ans[len], arr[i]);
}

// Some entries in ans[] may not be filled yet. Fill
// them by taking values from right side of ans[]
for (int i=n-1; i>=1; i--)
    ans[i] = Math.max(ans[i], ans[i+1]);
```

```
        // Print the result
        for (int i=1; i<=n; i++)
            System.out.print(ans[i] + " ");
    }

    // Driver method
    public static void main(String[] args)
    {
        printMaxOfMin(arr.length);
    }
}
```

Output:

70 30 20 10 10 10 10

Time Complexity: $O(n)$

Auxiliary Space: $O(n)$

This article is contributed by [Ekta Goel](#) and [Ayush Govil](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : [Sam007](#), [nitin mittal](#)

Source

<https://www.geeksforgeeks.org/find-the-maximum-of-minimums-for-every-window-size-in-a-given-array/>

Chapter 39

Find maximum sum possible equal sum of three stacks

Find maximum sum possible equal sum of three stacks - GeeksforGeeks

Given three stack of the positive numbers, the task is to find the possible equal maximum sum of the stacks with removal of top elements allowed. Stacks are represented as array, and the first index of the array represent the top element of the stack.

Examples:

```
Input : stack1[] = { 3, 10}  
        stack2[] = { 4, 5 }  
        stack3[] = { 2, 1 }
```

```
Output : 0
```

```
Sum can only be equal after removing all elements  
from all stacks.
```

Input	Output
<div style="border: 1px solid black; padding: 5px; display: inline-block;"> <div style="text-align: center;">3</div> <div style="text-align: center;">2</div> <div style="text-align: center;">1</div> <div style="text-align: center;">1</div> <div style="text-align: center;">1</div> </div>	<p>By removing 3 of stack1, sum of stack1 =</p> <p style="text-align: center;">$8 - 3 = 5$</p>
<div style="border: 1px solid black; padding: 5px; display: inline-block;"> <div style="text-align: center;">4</div> <div style="text-align: center;">3</div> <div style="text-align: center;">2</div> </div>	<p>By removing 4 of stack2, sum of stack2 =</p> <p style="text-align: center;">$9 - 4 = 5$</p>
<div style="border: 1px solid black; padding: 5px; display: inline-block;"> <div style="text-align: center;">2</div> <div style="text-align: center;">5</div> <div style="text-align: center;">4</div> <div style="text-align: center;">1</div> </div>	<p>By removing 2 and 5 of stack3, sum of stack3 =</p> <p style="text-align: center;">$12 - 5 - 2 = 5$</p>

The idea is to compare the sum of each stack, if they are not same, remove the top element of the stack having the maximum sum.

Algorithm for solving this problem:

1. Find sum of all elements of in individual stacks.
2. If the sum of all three stacks is same, then this is the maximum sum.
3. Else remove the top element of the stack having the maximum sum among three of stacks. Repeat step 1 and step 2.

The approach works because elements are positive. To make sum equal, we must remove some element from stack having more sum and we can only remove from top.

Below is the implementation of this approach:

C/C++

```
// C++ program to calculate maximum sum with equal
// stack sum.
#include<bits/stdc++.h>
using namespace std;
```

```
// Returns maximum possible equal sum of three stacks
// with removal of top elements allowed
int maxSum(int stack1[], int stack2[], int stack3[],
           int n1, int n2, int n3)
{
    int sum1 = 0, sum2 = 0, sum3 = 0;

    // Finding the initial sum of stack1.
    for (int i=0; i < n1; i++)
        sum1 += stack1[i];

    // Finding the initial sum of stack2.
    for (int i=0; i < n2; i++)
        sum2 += stack2[i];

    // Finding the initial sum of stack3.
    for (int i=0; i < n3; i++)
        sum3 += stack3[i];

    // As given in question, first element is top
    // of stack..
    int top1 =0, top2 = 0, top3 = 0;
    int ans = 0;
    while (1)
    {
        // If any stack is empty
        if (top1 == n1 || top2 == n2 || top3 == n3)
            return 0;

        // If sum of all three stack are equal.
        if (sum1 == sum2 && sum2 == sum3)
            return sum1;

        // Finding the stack with maximum sum and
        // removing its top element.
        if (sum1 >= sum2 && sum1 >= sum3)
            sum1 -= stack1[top1++];
        else if (sum2 >= sum3 && sum2 >= sum3)
            sum2 -= stack2[top2++];
        else if (sum3 >= sum2 && sum3 >= sum1)
            sum3 -= stack3[top3++];
    }
}

// Driven Program
int main()
{
```



```
int stack1[] = { 3, 2, 1, 1, 1 };
int stack2[] = { 4, 3, 2 };
int stack3[] = { 1, 1, 4, 1 };

int n1 = sizeof(stack1)/sizeof(stack1[0]);
int n2 = sizeof(stack2)/sizeof(stack2[0]);
int n3 = sizeof(stack3)/sizeof(stack3[0]);

cout << maxSum(stack1, stack2, stack3, n1, n2, n3) << endl;
return 0;
}
```

Java

```
// JAVA Code for Find maximum sum possible
// equal sum of three stacks
class GFG {

    // Returns maximum possible equal sum of three
    // stacks with removal of top elements allowed
    public static int maxSum(int stack1[], int stack2[],
                             int stack3[], int n1, int n2,
                             int n3)
    {
        int sum1 = 0, sum2 = 0, sum3 = 0;

        // Finding the initial sum of stack1.
        for (int i=0; i < n1; i++)
            sum1 += stack1[i];

        // Finding the initial sum of stack2.
        for (int i=0; i < n2; i++)
            sum2 += stack2[i];

        // Finding the initial sum of stack3.
        for (int i=0; i < n3; i++)
            sum3 += stack3[i];

        // As given in question, first element is top
        // of stack..
        int top1 =0, top2 = 0, top3 = 0;
        int ans = 0;
        while (true)
        {
            // If any stack is empty
            if (top1 == n1 || top2 == n2 || top3 == n3)
                return 0;
        }
    }
}
```

```
// If sum of all three stack are equal.
if (sum1 == sum2 && sum2 == sum3)
    return sum1;

// Finding the stack with maximum sum and
// removing its top element.
if (sum1 >= sum2 && sum1 >= sum3)
    sum1 -= stack1[top1++];
else if (sum2 >= sum3 && sum2 >= sum3)
    sum2 -= stack2[top2++];
else if (sum3 >= sum2 && sum3 >= sum1)
    sum3 -= stack3[top3++];
}
}

/* Driver program to test above function */
public static void main(String[] args)
{
    int stack1[] = { 3, 2, 1, 1, 1 };
    int stack2[] = { 4, 3, 2 };
    int stack3[] = { 1, 1, 4, 1 };

    int n1 = stack1.length;
    int n2 = stack2.length;
    int n3 = stack3.length;

    System.out.println(maxSum(stack1, stack2,
                               stack3, n1, n2, n3));
}
}
// This code is contributed by Arnav Kr. Mandal.
```

Python

```
# Python program to calculate maximum sum with equal
# stack sum.
# Returns maximum possible equal sum of three stacks
# with removal of top elements allowed
def maxSum(stack1, stack2, stack3, n1, n2, n3):
    sum1, sum2, sum3 = 0, 0, 0

    # Finding the initial sum of stack1.
    for i in range(n1):
        sum1 += stack1[i]

    # Finding the initial sum of stack2.
    for i in range(n2):
        sum2 += stack2[i]
```

```
# Finding the initial sum of stack3.
for i in range(n3):
    sum3 += stack3[i]

# As given in question, first element is top
# of stack..
top1, top2, top3 = 0, 0, 0
ans = 0
while (1):
    # If any stack is empty
    if (top1 == n1 or top2 == n2 or top3 == n3):
        return 0

    # If sum of all three stack are equal.
    if (sum1 == sum2 and sum2 == sum3):
        return sum1

    # Finding the stack with maximum sum and
    # removing its top element.
    if (sum1 >= sum2 and sum1 >= sum3):
        sum1 -= stack1[top1]
        top1=top1+1
    elif (sum2 >= sum3 and sum2 >= sum3):
        sum2 -= stack2[top2]
        top2=top2+1
    elif (sum3 >= sum2 and sum3 >= sum1):
        sum3 -= stack3[top3]
        top3=top3+1

# Driven Program
stack1 = [ 3, 2, 1, 1, 1 ]
stack2 = [ 4, 3, 2 ]
stack3 = [ 1, 1, 4, 1 ]

n1 = len(stack1)
n2 = len(stack2)
n3 = len(stack3)

print maxSum(stack1, stack2, stack3, n1, n2, n3)

#This code is contributed by Afzal Ansari
```

C#

```
// C# Code for Find maximum sum with
// equal sum of three stacks
using System;
```

```
class GFG {

    // Returns maximum possible equal
    // sum of three stacks with removal
    // of top elements allowed
    public static int maxSum(int[] stack1,
        int[] stack2, int[] stack3,
        int n1, int n2, int n3)
    {

        int sum1 = 0, sum2 = 0, sum3 = 0;

        // Finding the initial sum of
        // stack1.
        for (int i = 0; i < n1; i++)
            sum1 += stack1[i];

        // Finding the initial sum of
        // stack2.
        for (int i = 0; i < n2; i++)
            sum2 += stack2[i];

        // Finding the initial sum of
        // stack3.
        for (int i = 0; i < n3; i++)
            sum3 += stack3[i];

        // As given in question, first
        // element is top of stack..
        int top1 = 0, top2 = 0, top3 = 0;

        while (true) {

            // If any stack is empty
            if (top1 == n1 || top2 == n2
                || top3 == n3)
                return 0;

            // If sum of all three stack
            // are equal.
            if (sum1 == sum2 && sum2 == sum3)
                return sum1;

            // Finding the stack with maximum
            // sum and removing its top element.
            if (sum1 >= sum2 && sum1 >= sum3)
                sum1 -= stack1[top1++];
```

```
        else if (sum2 >= sum3 && sum2 >= sum3)
            sum2 -= stack2[top2++];
        else if (sum3 >= sum2 && sum3 >= sum1)
            sum3 -= stack3[top3++];
    }
}

/* Driver program to test above function */
public static void Main()
{
    int[] stack1 = { 3, 2, 1, 1, 1 };
    int[] stack2 = { 4, 3, 2 };
    int[] stack3 = { 1, 1, 4, 1 };

    int n1 = stack1.Length;
    int n2 = stack2.Length;
    int n3 = stack3.Length;

    Console.WriteLine(maxSum(stack1, stack2,
                             stack3, n1, n2, n3));
}

// This code is contributed by nitin mittal.
```

PHP

```
<?php
// PHP program to calculate maximum
// sum with equal stack sum.

// Returns maximum possible
// equal sum of three stacks
// with removal of top elements
// allowed
function maxSum($stack1, $stack2, $stack3,
                $n1, $n2, $n3)
{
    $sum1 = 0; $sum2 = 0; $sum3 = 0;

    // Finding the initial sum of stack1.
    for ($i = 0; $i < $n1; $i++)
        $sum1 += $stack1[$i];

    // Finding the initial sum of stack2.
    for ($i = 0; $i < $n2; $i++)
        $sum2 += $stack2[$i];
```

```
// Finding the initial sum of stack3.
for ($i = 0; $i < $n3; $i++)
    $sum3 += $stack3[$i];

// As given in question,
// first element is top
// of stack..
$top1 = 0;
$top2 = 0;
$top3 = 0;
$ans = 0;
while (1)
{
    // If any stack is empty
    if ($top1 == $n1 || $top2 == $n2 ||
        $top3 == $n3)
        return 0;

    // If sum of all three stack are equal.
    if ($sum1 == $sum2 && $sum2 == $sum3)
        return $sum1;

    // Finding the stack with
    // maximum sum and
    // removing its top element.
    if ($sum1 >= $sum2 && $sum1 >= $sum3)
        $sum1 -= $stack1[$top1++];

    else if ($sum2 >= $sum3 && $sum2 >= $sum3)
        $sum2 -= $stack2[$top2++];

    else if ($sum3 >= $sum2 && $sum3 >= $sum1)
        $sum3 -= $stack3[$top3++];
}

}

// Driver Code
$stack1 = array(3, 2, 1, 1, 1);
$stack2 = array(4, 3, 2);
$stack3 = array(1, 1, 4, 1);

$n1 = sizeof($stack1);
$n2 = sizeof($stack2);
$n3 = sizeof($stack3);
echo maxSum($stack1, $stack2,
            $stack3, $n1,
            $n2, $n3) ;
```

```
// This code is contributed by nitin mittal  
?>
```

Output:

5

Time Complexity : $O(n_1 + n_2 + n_3)$ where n_1 , n_2 and n_3 are sizes of three stacks.

Improved By : [nitin mittal](#)

Source

<https://www.geeksforgeeks.org/find-maximum-sum-possible-equal-sum-three-stacks/>

Chapter 40

Find next Smaller of next Greater in an array

Find next Smaller of next Greater in an array - GeeksforGeeks

Given array of integer, find the next smaller of [next greater element](#) of every element in array.

Note : Elements for which no greater element exists or no smaller of greater element exist, print -1.

Examples:

Input : arr[] = {5, 1, 9, 2, 5, 1, 7}

Output: 2 2 -1 1 -1 -1 -1

Explanation :

Next Greater ->	Right Smaller
5 -> 9	9 -> 2
1 -> 9	9 -> 2
9 -> -1	-1 -> -1
2 -> 5	5 -> 1
5 -> 7	7 -> -1
1 -> 7	7 -> -1
7 -> -1	-1 -> -1

Input : arr[] = {4, 8, 2, 1, 9, 5, 6, 3}

Output : 2 5 5 5 -1 3 -1 -1

A **simple solution** is to iterate through all elements. For every element, find the next greater element of current element and then find right smaller element for current next greater element. Time taken of this solution is $O(n^2)$.

An **efficient solution** takes $O(n)$ time. Notice that it is the combination of [Next greater element](#) & [next smaller element](#) in array.

Let input array be 'arr[]' and size of array be 'n'
find next greatest element of every element

step 1 : Create an empty stack (S) in which we store the indexes
and NG[] that is user to store the indexes of NGE
of every element.

step 2 : Traverse the array in reverse order
where i goes from (n-1 to 0)

- a) While S is nonempty and the top element of
S is smaller than or equal to 'arr[i]':
pop S
- b) If S is empty
arr[i] has no greater element
NG[i] = -1
- c) else we have next greater element
NG[i] = S.top() // here we store the index of NGE
- d). push current element index in stack
S.push(i)

Find Right smaller element of every element

step 3 : create an array RS[] used to store the index of
right smallest element

step 4 : we repeat step (1 & 2) with little bit of
modification in step 1 & 2 .
they are :

- a). we use RS[] in place of NG[].
- b). In step (2.a)
we pop element form stack S while S is not
empty or the top element of S is greater then
or equal to 'arr[i]'

step 5 . compute all RSE of NGE :

```
where i goes from 0 to n-1
if NG[ i ] != -1 && RS[ NG [ i]] != -1
    print arr[RS[NG[i]]]
else
    print -1
```

Below is C++ implementation of above idea

C++

```
// C++ Program to find Right smaller element of next
// greater element
#include<bits/stdc++.h>
using namespace std;

// function find Next greater element
void nextGreater(int arr[], int n, int next[], char order)
{
    // create empty stack
    stack<int> S;

    // Traverse all array elements in reverse order
    // order == 'G' we compute next greater elements of
    // every element
    // order == 'S' we compute right smaller element of
    // every element
    for (int i=n-1; i>=0; i--)
    {
        // Keep removing top element from S while the top
        // element is smaller then or equal to arr[i] (if Key is G)
        // element is greater then or equal to arr[i] (if order is S)
        while (!S.empty() &&
            ((order=='G')? arr[S.top()] <= arr[i]:
            arr[S.top()] >= arr[i]))
            S.pop();

        // store the next greater element of current element
        if (!S.empty())
            next[i] = S.top();

        // If all elements in S were smaller than arr[i]
        else
            next[i] = -1;

        // Push this element
        S.push(i);
    }
}

// Function to find Right smaller element of next greater
// element
void nextSmallerOfNextGreater(int arr[], int n)
```

```
{
    int NG[n]; // stores indexes of next greater elements
    int RS[n]; // stores indexes of right smaller elements

    // Find next greater element
    // Here G indicate next greater element
    nextGreater(arr, n, NG, 'G');

    // Find right smaller element
    // using same function nextGreater()
    // Here S indicate right smaller elements
    nextGreater(arr, n, RS, 'S');

    // If NG[i] == -1 then there is no smaller element
    // on right side. We can find Right smaller of next
    // greater by arr[RS[NG[i]]]
    for (int i=0; i< n; i++)
    {
        if (NG[i] != -1 && RS[NG[i]] != -1)
            cout << arr[RS[NG[i]]] << " ";
        else
            cout<<"-1"<<" ";
    }
}

// Driver program
int main()
{
    int arr[] = {5, 1, 9, 2, 5, 1, 7};
    int n = sizeof(arr)/sizeof(arr[0]);
    nextSmallerOfNextGreater(arr, n);
    return 0;
}
```

Python 3

```
# Python 3 Program to find Right smaller element of next
# greater element

# function find Next greater element
def nextGreater(arr, n, next, order):

    S = []

    # Traverse all array elements in reverse order
    # order == 'G' we compute next greater elements of
    # every element
    # order == 'S' we compute right smaller element of
```

```
#                every element
for i in range(n-1,-1,-1):

    # Keep removing top element from S while the top
    # element is smaller then or equal to arr[i] (if Key is G)
    # element is greater then or equal to arr[i] (if order is S)
    while (S!=[] and (arr[S[len(S)-1]] <= arr[i]
    if (order=='G') else arr[S[len(S)-1]] >= arr[i] )):

        S.pop()

    # store the next greater element of current element
    if (S!=[]):
        next[i] = S[len(S)-1]

    # If all elements in S were smaller than arr[i]
    else:
        next[i] = -1

    # Push this element
    S.append(i)

# Function to find Right smaller element of next greater
# element
def nextSmallerOfNextGreater(arr, n):
    NG = [None]*n # stores indexes of next greater elements
    RS = [None]*n # stores indexes of right smaller elements

    # Find next greater element
    # Here G indicate next greater element
    nextGreater(arr, n, NG, 'G')

    # Find right smaller element
    # using same function nextGreater()
    # Here S indicate right smaller elements
    nextGreater(arr, n, RS, 'S')

    # If NG[i] == -1 then there is no smaller element
    # on right side. We can find Right smaller of next
    # greater by arr[RS[NG[i]]]
    for i in range(n):
        if (NG[i] != -1 and RS[NG[i]] != -1):
            print(arr[RS[NG[i]]],end=" ")
        else:
            print("-1",end=" ")

# Driver program
if __name__=="__main__":
```

```
arr = [5, 1, 9, 2, 5, 1, 7]
n = len(arr)
nextSmallerOfNextGreater(arr, n)
```

this code is contributed by ChitraNayal

Output:

2 2 -1 1 -1 -1 -1

Time complexity : $O(n)$

Improved By : [ChitraNayal](#)

Source

<https://www.geeksforgeeks.org/find-next-smaller-next-greater-array/>

Chapter 41

Find the nearest smaller numbers on left side in an array

Find the nearest smaller numbers on left side in an array - GeeksforGeeks

Given an array of integers, find the nearest smaller number for every element such that the smaller element is on left side.

Examples:

```
Input: arr[] = {1, 6, 4, 10, 2, 5}
Output:      {_, 1, 1,  4, 1, 2}
First element ('1') has no element on left side. For 6,
there is only one smaller element on left side '1'.
For 10, there are three smaller elements on left side (1,
6 and 4), nearest among the three elements is 4.
```

```
Input: arr[] = {1, 3, 0, 2, 5}
Output:      {_, 1, _, 0, 2}
```

Expected time complexity is $O(n)$.

A **Simple Solution** is to use two nested loops. The outer loop starts from second element, the inner loop goes to all elements on left side of the element picked by outer loop and stops as soon as it finds a smaller element.

C++

```
// C++ implementation of simple algorithm to find
// smaller element on left side
#include <iostream>
using namespace std;
```

```
// Prints smaller elements on left side of every element
void printPrevSmaller(int arr[], int n)
{
    // Always print empty or '_' for first element
    cout << "_, ";

    // Start from second element
    for (int i=1; i<n; i++)
    {
        // look for smaller element on left of 'i'
        int j;
        for (j=i-1; j>=0; j--)
        {
            if (arr[j] < arr[i])
            {
                cout << arr[j] << ", ";
                break;
            }
        }

        // If there is no smaller element on left of 'i'
        if (j == -1)
            cout << "_, " ;
    }
}

/* Driver program to test insertion sort */
int main()
{
    int arr[] = {1, 3, 0, 2, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    printPrevSmaller(arr, n);
    return 0;
}
```

Java

```
// Java implementation of simple
// algorithm to find smaller
// element on left side
import java.io.*;
class GFG {

    // Prints smaller elements on
    // left side of every element
    static void printPrevSmaller(int []arr, int n)
    {
```

```
// Always print empty or '_'
// for first element
System.out.print( "_", );

// Start from second element
for (int i = 1; i < n; i++)
{
    // look for smaller
    // element on left of 'i'
    int j;
    for(j = i - 1; j >= 0; j--)
    {
        if (arr[j] < arr[i])
        {
            System.out.print(arr[j] + ", ");
            break;
        }
    }

    // If there is no smaller
    // element on left of 'i'
    if (j == -1)
        System.out.print( "_", );
}

// Driver Code
public static void main (String[] args)
{
    int []arr = {1, 3, 0, 2, 5};
    int n = arr.length;
    printPrevSmaller(arr, n);
}

// This code is contributed by anuj_67.
```

Python3

```
# Python 3 implementation of simple
# algorithm to find smaller element
# on left side

# Prints smaller elements on left
# side of every element
def printPrevSmaller(arr, n):
```



```
# Always print empty or '_' for
# first element
print("_", ", end=")

# Start from second element
for i in range(1, n ):

    # look for smaller element
    # on left of 'i'
    for j in range(i-1 ,-2 ,-1):

        if (arr[j] < arr[i]):

            print(arr[j] ,", ",
                  end=")

            break

    # If there is no smaller
    # element on left of 'i'
    if (j == -1):
        print("_", ", end=")

# Driver program to test insertion
# sort
arr = [1, 3, 0, 2, 5]
n = len(arr)
printPrevSmaller(arr, n)

# This code is contributed by
# Smitha
```

C#

```
// C# implementation of simple
// algorithm to find smaller
// element on left side
using System;

class GFG {

    // Prints smaller elements on
    // left side of every element
    static void printPrevSmaller(int []arr,
                                  int n)
    {

        // Always print empty or '_'
        // for first element
```

```
Console.Write( "_", " );

// Start from second element
for (int i = 1; i < n; i++)
{
    // look for smaller
    // element on left of 'i'
    int j;
    for(j = i - 1; j >= 0; j--)
    {
        if (arr[j] < arr[i])
        {
            Console.Write(arr[j]
                           + ", ");
            break;
        }
    }

    // If there is no smaller
    // element on left of 'i'
    if (j == -1)
        Console.Write( "_", " );
}

// Driver Code
public static void Main ()
{
    int []arr = {1, 3, 0, 2, 5};
    int n = arr.Length;
    printPrevSmaller(arr, n);
}

// This code is contributed by anuj_67.
```

PHP

```
<?php
// PHP implementation of simple
// algorithm to find smaller
// element on left side

// Prints smaller elements on
// left side of every element
function printPrevSmaller( $arr, $n)
{
```

```
// Always print empty or
// '_' for first element
echo "_, ";

// Start from second element
for($i = 1; $i < $n; $i++)
{
    // look for smaller
    // element on left of 'i'
    $j;
    for($j = $i - 1; $j >= 0; $j--)
    {
        if ($arr[$j] < $arr[$i])
        {
            echo $arr[$j] , ", ";
            break;
        }
    }

    // If there is no smaller
    // element on left of 'i'
    if ($j == -1)
        echo "_, " ;
    }
}

// Driver Code
$arr = array(1, 3, 0, 2, 5);
$n = count($arr);
printPrevSmaller($arr, $n);

// This code is contributed by anuj_67.
?>
```

Output:

_, 1, _, 0, 2, ,

Time complexity of the above solution is $O(n^2)$.

There can be an **Efficient Solution** that works in $O(n)$ time. The idea is to use a stack. Stack is used to maintain a subsequence of the values that have been processed so far and are smaller than any later value that has already been processed.

Below is stack based algorithm

Let input sequence be 'arr[]' and size of array be 'n'

- 1) Create a new empty stack S
- 2) For every element 'arr[i]' in the input sequence 'arr[]', where 'i' goes from 0 to n-1.
 - a) while S is nonempty and the top element of S is greater than or equal to 'arr[i]':
 pop S
 - b) if S is empty:
 'arr[i]' has no preceding smaller value
 - c) else:
 the nearest smaller value to 'arr[i]' is
 the top element of S
 - d) push 'arr[i]' onto S

Below is C++ implementation of above algorithm.

```
// C++ implementation of simple algorithm to find
// smaller element on left side
#include <iostream>
#include <stack>
using namespace std;

// Prints smaller elements on left side of every element
void printPrevSmaller(int arr[], int n)
{
    // Create an empty stack
    stack<int> S;

    // Traverse all array elements
    for (int i=0; i<n; i++)
    {
        // Keep removing top element from S while the top
        // element is greater than or equal to arr[i]
        while (!S.empty() && S.top() >= arr[i])
            S.pop();

        // If all elements in S were greater than arr[i]
        if (S.empty())
            cout << "_, ";
        else //Else print the nearest smaller element
            cout << S.top() << ", ";

        // Push this element
        S.push(arr[i]);
    }
}
```

```
    }  
}  
  
/* Driver program to test insertion sort */  
int main()  
{  
    int arr[] = {1, 3, 0, 2, 5};  
    int n = sizeof(arr)/sizeof(arr[0]);  
    printPrevSmaller(arr, n);  
    return 0;  
}
```

Output:

_, 1, _, 0, 2,

Time complexity of above program is $O(n)$ as every element is pushed and popped at most once to the stack. So overall constant number of operations are performed per element.

This article is contributed by Ashish Kumar Singh. Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

Improved By : [vt_m](#), [Smitha Dinesh Semwal](#)

Source

<https://www.geeksforgeeks.org/find-the-nearest-smaller-numbers-on-left-side-in-an-array/>

Chapter 42

Form minimum number from given sequence

Form minimum number from given sequence - GeeksforGeeks

Given a pattern containing only I's and D's. **I** for increasing and **D** for decreasing. Devise an algorithm to print the minimum number following that pattern. Digits from 1-9 and digits can't repeat.

Examples:

Input: D	Output: 21
Input: I	Output: 12
Input: DD	Output: 321
Input: II	Output: 123
Input: DIDI	Output: 21435
Input: IIDDD	Output: 126543
Input: DDIDDIID	Output: 321654798

Source: [Amazon Interview Question](#)

Below are some important observations

Since digits can't repeat, there can be at most 9 digits in output.

Also number of digits in output is one more than number of characters in input. Note that the first character of input corresponds to two digits in output.

Idea is to iterate over input array and keep track of last printed digit and maximum digit printed so far. Below is the implementation of above idea.

C++

```
// C++ program to print minimum number that can be formed
```

```
// from a given sequence of Is and Ds
#include <iostream>
using namespace std;

// Prints the minimum number that can be formed from
// input sequence of I's and D's
void PrintMinNumberForPattern(string arr)
{
    // Initialize current_max (to make sure that
    // we don't use repeated character
    int curr_max = 0;

    // Initialize last_entry (Keeps track for
    // last printed digit)
    int last_entry = 0;

    int j;

    // Iterate over input array
    for (int i=0; i<arr.length(); i++)
    {
        // Initialize 'noOfNextD' to get count of
        // next D's available
        int noOfNextD = 0;

        switch(arr[i])
        {
            case 'I':
                // If letter is 'I'

                // Calculate number of next consecutive D's
                // available
                j = i+1;
                while (arr[j] == 'D' && j < arr.length())
                {
                    noOfNextD++;
                    j++;
                }

                if (i==0)
                {
                    curr_max = noOfNextD + 2;

                    // If 'I' is first letter, print incremented
                    // sequence from 1
                    cout << " " << ++last_entry;
                    cout << " " << curr_max;
```

```
        // Set max digit reached
        last_entry = curr_max;
    }
    else
    {
        // If not first letter

        // Get next digit to print
        curr_max = curr_max + noOfNextD + 1;

        // Print digit for I
        last_entry = curr_max;
        cout << " " << last_entry;
    }

    // For all next consecutive 'D' print
    // decremented sequence
    for (int k=0; k<noOfNextD; k++)
    {
        cout << " " << --last_entry;
        i++;
    }
    break;

// If letter is 'D'
case 'D':
    if (i == 0)
    {
        // If 'D' is first letter in sequence
        // Find number of Next D's available
        j = i+1;
        while (arr[j] == 'D' && j < arr.length())
        {
            noOfNextD++;
            j++;
        }

        // Calculate first digit to print based on
        // number of consecutive D's
        curr_max = noOfNextD + 2;

        // Print twice for the first time
        cout << " " << curr_max << " " << curr_max - 1;

        // Store last entry
        last_entry = curr_max - 1;
    }
    else
```



```
        {
            // If current 'D' is not first letter

            // Decrement last_entry
            cout << " " << last_entry - 1;
            last_entry--;
        }
        break;
    }
}
cout << endl;
}

// Driver program to test above
int main()
{
    PrintMinNumberForPattern("IDID");
    PrintMinNumberForPattern("I");
    PrintMinNumberForPattern("DD");
    PrintMinNumberForPattern("II");
    PrintMinNumberForPattern("DIDI");
    PrintMinNumberForPattern("IIDDD");
    PrintMinNumberForPattern("DDIDDIID");
    return 0;
}
```

PHP

```
<?php
// PHP program to print minimum
// number that can be formed
// from a given sequence of
// Is and Ds

// Prints the minimum number
// that can be formed from
// input sequence of I's and D's
function PrintMinNumberForPattern($arr)
{
    // Initialize current_max
    // (to make sure that
    // we don't use repeated
    // character
    $curr_max = 0;

    // Initialize last_entry
    // (Keeps track for
    // last printed digit)
```

```
$last_entry = 0;

$j;

// Iterate over
// input array
for ($i = 0; $i < strlen($arr); $i++)
{
    // Initialize 'noOfNextD'
    // to get count of
    // next D's available
    $noOfNextD = 0;

    switch($arr[$i])
    {
        case 'I':
            // If letter is 'I'

            // Calculate number of
            // next consecutive D's
            // available
            $j = $i + 1;
            while ($arr[$j] == 'D' &&
                $j < strlen($arr))
            {
                $noOfNextD++;
                $j++;
            }

            if ($i == 0)
            {
                $curr_max = $noOfNextD + 2;

                // If 'I' is first letter,
                // print incremented
                // sequence from 1
                echo " " , ++$last_entry;
                echo " " , $curr_max;

                // Set max
                // digit reached
                $last_entry = $curr_max;
            }
            else
            {
                // If not first letter

                // Get next digit
```

```
// to print
$curr_max = $curr_max +
    $noOfNextD + 1;

// Print digit for I
$last_entry = $curr_max;
echo " " , $last_entry;
}

// For all next consecutive 'D'
// print decremented sequence
for ($k = 0; $k < $noOfNextD; $k++)
{
    echo " " , --$last_entry;
    $i++;
}
break;

// If letter is 'D'
case 'D':
    if ($i == 0)
    {
        // If 'D' is first letter
        // in sequence. Find number
        // of Next D's available
        $j = $i+1;
        while (($arr[$j] == 'D') &&
            ($j < strlen($arr)))
        {
            $noOfNextD++;
            $j++;
        }

        // Calculate first digit
        // to print based on
        // number of consecutive D's
        $curr_max = $noOfNextD + 2;

        // Print twice for
        // the first time
        echo " " , $curr_max ,
            " " , $curr_max - 1;

        // Store last entry
        $last_entry = $curr_max - 1;
    }
    else
    {
```

```
        // If current 'D'
        // is not first letter

        // Decrement last_entry
        echo " " , $last_entry - 1;
        $last_entry--;
    }
    break;
}

echo "\n";
}

// Driver Code
PrintMinNumberForPattern("IDID");
PrintMinNumberForPattern("I");
PrintMinNumberForPattern("DD");
PrintMinNumberForPattern("II");
PrintMinNumberForPattern("DIDI");
PrintMinNumberForPattern("IIDDD");
PrintMinNumberForPattern("DDIDDIID");

// This code is contributed by aj_36
?>
```

Output:

```
1 3 2 5 4
1 2
3 2 1
1 2 3
2 1 4 3 5
1 2 6 5 4 3
3 2 1 6 5 4 7 9 8
```

This solution is suggested by Swapnil Trambake.

Alternate Solution:

Let's observe a few facts in case of minimum number:

- The digits can't repeat hence there can be 9 digits at most in output.
- To form a minimum number , at every index of the output, we are interested in the minimum number which can be placed at that index.

The idea is to iterate over the entire input array , keeping track of the minimum number (1-9) which can be placed at that position of the output.

The tricky part of course occurs when 'D' is encountered at index other than 0. In such a case we have to track the nearest 'I' to the left of 'D' and increment each number in the output vector by 1 in between 'I' and 'D'.

We cover the base case as follows:

- If the first character of input is 'I' then we append 1 and 2 in the output vector and the minimum available number is set to 3. The index of most recent 'I' is set to 1.
- If the first character of input is 'D' then we append 2 and 1 in the output vector and the minimum available number is set to 3, and the index of most recent 'I' is set to 0.

Now we iterate the input string from index 1 till its end and:

- If the character scanned is 'I', minimum value which has not been used yet is appended to the output vector. We increment the value of minimum no. available and index of most recent 'I' is also updated.
- If the character scanned is 'D' at index *i* of input array, we append the *i*th element from output vector in the output and track the nearest 'I' to the left of 'D' and increment each number in the output vector by 1 in between 'I' and 'D'.

Following is the program for the same:

C++

```
// C++ program to print minimum number that can be formed
// from a given sequence of Is and Ds
#include<bits/stdc++.h>
using namespace std;

void printLeast(string arr)
{
    // min_avail represents the minimum number which is
    // still available for inserting in the output vector.
    // pos_of_I keeps track of the most recent index
    // where 'I' was encountered w.r.t the output vector
    int min_avail = 1, pos_of_I = 0;

    //vector to store the output
    vector<int>v;

    // cover the base cases
    if (arr[0]=='I')
    {
        v.push_back(1);
        v.push_back(2);
        min_avail = 3;
        pos_of_I = 1;
    }
```

```
    }
    else
    {
        v.push_back(2);
        v.push_back(1);
        min_avail = 3;
        pos_of_I = 0;
    }

    // Traverse rest of the input
    for (int i=1; i<arr.length(); i++)
    {
        if (arr[i]=='I')
        {
            v.push_back(min_avail);
            min_avail++;
            pos_of_I = i+1;
        }
        else
        {
            v.push_back(v[i]);
            for (int j=pos_of_I; j<=i; j++)
                v[j]++;

            min_avail++;
        }
    }

    // print the number
    for (int i=0; i<v.size(); i++)
        cout << v[i] << " ";
    cout << endl;
}

// Driver program to check the above function
int main()
{
    printLeast("IDID");
    printLeast("I");
    printLeast("DD");
    printLeast("II");
    printLeast("DIDI");
    printLeast("IIDDD");
    printLeast("DDIDDIID");
    return 0;
}
```

Output:

```
1 3 2 5 4
1 2
3 2 1
1 2 3
2 1 4 3 5
1 2 6 5 4 3
3 2 1 6 5 4 7 9 8
```

This solution is suggested by [Ashutosh Kumar](#).

Method 3

We can that when we encounter 'I', we got numbers in increasing order but if we encounter 'D', we want to have numbers in decreasing order. Length of the output string is always one more than the input string. So loop is from 0 till the length of the string. We have to take numbers from 1-9 so we always push (i+1) to our stack. Then we check what is the resulting character at the specified index. So, there will be two cases which are as follows:-

Case 1: If we have encountered 'I' or we are at the last character of input string, then pop from the stack and add it to the end of the output string until the stack gets empty.

Case 2: If we have encountered 'D', then we want the numbers in decreasing order. So we just push (i+1) to our stack.

C++

```
#include <bits/stdc++.h>
using namespace std;

// Function to decode the given sequence to construct
// minimum number without repeated digits
void PrintMinNumberForPattern(string seq)
{
    // result store output string
    string result;

    // create an empty stack of integers
    stack<int> stk;

    // run n+1 times where n is length of input sequence
    for (int i = 0; i <= seq.length(); i++)
    {
        // push number i+1 into the stack
        stk.push(i + 1);

        // if all characters of the input sequence are
        // processed or current character is 'I'
        // (increasing)
        if (i == seq.length() || seq[i] == 'I')
        {
            while (!stk.empty())
            {
                result += stk.top();
                stk.pop();
            }
        }
    }
}
```

```
        {
            // run till stack is empty
            while (!stk.empty())
            {
                // remove top element from the stack and
                // add it to solution
                result += to_string(stk.top());
                result += " ";
                stk.pop();
            }
        }

        cout << result << endl;
    }

// main function
int main()
{
    PrintMinNumberForPattern("IDID");
    PrintMinNumberForPattern("I");
    PrintMinNumberForPattern("DD");
    PrintMinNumberForPattern("II");
    PrintMinNumberForPattern("DIDI");
    PrintMinNumberForPattern("IIDDD");
    PrintMinNumberForPattern("DDIDDIID");
    return 0;
}
```

Output:

```
1 3 2 5 4
1 2
3 2 1
1 2 3
2 1 4 3 5
1 2 6 5 4 3
3 2 1 6 5 4 7 9 8
```

Time Complexity : $O(n)$

Auxiliary Space : $O(n)$

This method is contributed by **Roshni Agarwal**.

Improved By : [jit_t](#)

Source

<https://www.geeksforgeeks.org/form-minimum-number-from-given-sequence/>

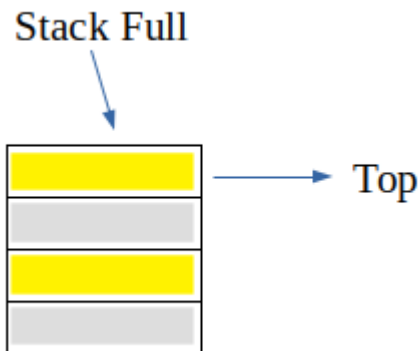
Chapter 43

Growable array based stack

Growable array based stack - GeeksforGeeks

We all know about [Stacks](#) also known as **Last-In-First-Out(LIFO)** structures. Stack primarily has two main operation namely push and pop, where push inserts an element at top and pop removes an element from top of the stack.

Now, whenever an implementation of stack is considered its size is pre-determined or fixed. Even though it is dynamically allocated, still once it is made its size cannot be changed. And hence a condition called “stack full” arises.



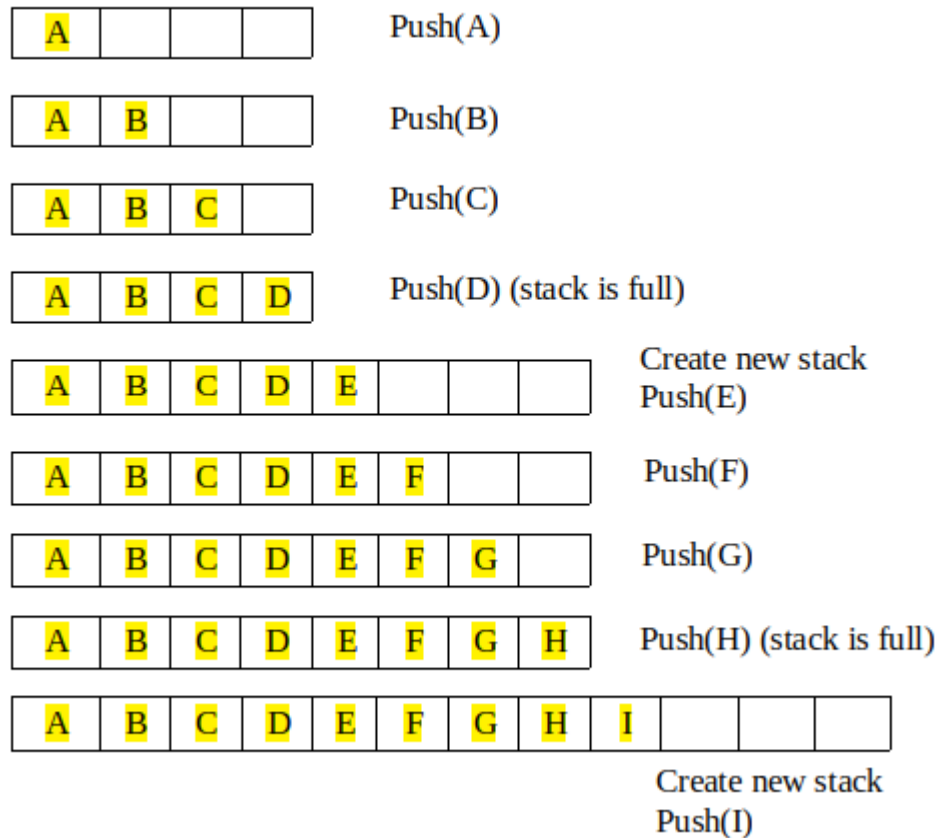
But what if a stack can grow as more elements are inserted or more elements are going to be inserted in future. Remember, we are talking about array based Stack. **Growable Stack** is the concept of allocating more memory such that “stack full” condition does not arise easily.

A Growable array-based Stack can be implemented by allocating new memory larger than previous stack memory and copying elements from old stack to new stack. And then at last change the name of new stack to the name which was given to old stack

There are two strategy for growable stack:

1. **Tight Strategy** : Add a constant amount to the old stack ($N+c$)
2. **Growth Strategy** : Double the size of old stack ($2N$)

TIGHT STRATEGY



There are two operation on growable stack:

1. Regular Push Operation: Add one element at top of stack
2. Special Push Operation: Create a new stack of size greater than old stack (according to one of the strategy above) and copy all elements from old stack and then push the new element to the new stack.

```
// CPP Program to implement growable array based stack
// using tight strategy
#include <iostream>
using namespace std;

// constant amount at which stack is increased
#define BOUND 4
```

```
// top of the stack
int top = -1;

// length of stack
int length = 0;

// function to create new stack
int* create_new(int* a)
{
    // allocate memory for new stack
    int* new_a = new int[length + BOUND];

    // copying the content of old stack
    for (int i = 0; i < length; i++)
        new_a[i] = a[i];

    // re-sizing the length
    length += BOUND;
    return new_a;
}

// function to push new element
int* push(int* a, int element)
{
    // if stack is full, create new one
    if (top == length - 1)
        a = create_new(a);

    // insert element at top of the stack
    a[++top] = element;
    return a;
}

// function to pop an element
void pop(int* a)
{
    top--;
}

// function to display
void display(int* a)
{
    // if top is -1, that means stack is empty
    if (top == -1)
        cout << "Stack is Empty" << endl;
    else {
        cout << "Stack: ";
        for (int i = 0; i <= top; i++)
```

```
        cout << a[i] << " ";
        cout << endl;
    }
}

// Driver Code
int main()
{
    // creating initial stack
    int *a = create_new(a);

    // pushing element to top of stack
    a = push(a, 1);
    a = push(a, 2);
    a = push(a, 3);
    a = push(a, 4);
    display(a);

    // pushing more element when stack is full
    a = push(a, 5);
    a = push(a, 6);
    display(a);

    a = push(a, 7);
    a = push(a, 8);
    display(a);

    // pushing more element so that stack can grow
    a = push(a, 9);
    display(a);

    return 0;
}
```

Output:

```
Stack: 1 2 3 4
Stack: 1 2 3 4 5 6
Stack: 1 2 3 4 5 6 7 8
Stack: 1 2 3 4 5 6 7 8 9
```

Source

<https://www.geeksforgeeks.org/growable-array-based-stack/>

Chapter 44

How to create mergable stack?

How to create mergable stack? - GeeksforGeeks

Design a stack with following operations.

- a) push(Stack s, x): Adds an item x to stack s
- b) pop(Stack s): Removes the top item from stack s
- c) merge(Stack s1, Stack s2): Merge contents of s2 into s1.

Time Complexity of all above operations should be $O(1)$.

If we **use array** implementation of stack, then merge is not possible to do in $O(1)$ time as we have to do following steps.

- a) Delete old arrays
- b) Create a new array for s1 with size equal to size of old array for s1 plus size of s2.
- c) Copy old contents of s1 and s2 to new array for s1

The above operations take $O(n)$ time.

We can **use a linked list** with two pointers, one pointer to first node (also used as top when elements are added and removed from beginning). The other pointer is needed for last node so that we can quickly link the linked list of s2 at the end of s1. Following are all operations.

- a) push(): Adds the new item at the beginning of linked list using first pointer.
- b) pop(): Removes an item from beginning using first pointer.
- c) merge(): Links the first pointer second stack as next of last pointer of first list.

Can we do it if we are not allowed to use extra pointer?

We can do it with **circular linked list**. The idea is to keep track of last node in linked list. The next of last node indicates top of stack.

- a) push(): Adds the new item as next of last node.
- b) pop(): Removes next of last node.
- c) merge(): Links the top (next of last) of second list to the top (next of last) of first list. And makes last of second list as last of whole list.

This article is contributed by **Rahul Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/create-mergable-stack/>

Chapter 45

How to efficiently implement k stacks in a single array?

How to efficiently implement k stacks in a single array? - GeeksforGeeks

We have discussed [space efficient implementation of 2 stacks in a single array](#). In this post, a general solution for k stacks is discussed. Following is the detailed problem statement.

Create a data structure $kStacks$ that represents k stacks. Implementation of $kStacks$ should use only one array, i.e., k stacks should use the same array for storing elements. Following functions must be supported by $kStacks$.

`push(int x, int sn) ->` pushes x to stack number 'sn' where sn is from 0 to k-1

`pop(int sn) ->` pops an element from stack number 'sn' where sn is from 0 to k-1

Method 1 (Divide the array in slots of size n/k)

A simple way to implement k stacks is to divide the array in k slots of size n/k each, and fix the slots for different stacks, i.e., use `arr[0]` to `arr[n/k-1]` for first stack, and `arr[n/k]` to `arr[2n/k-1]` for stack2 where `arr[]` is the array to be used to implement two stacks and size of array be n.

The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in `arr[]`. For example, say the k is 2 and array size (n) is 6 and we push 3 elements to first and do not push anything to second stack. When we push 4th element to first, there will be overflow even if we have space for 3 more elements in array.

Method 2 (A space efficient implementation)

The idea is to use two extra arrays for efficient implementation of k stacks in an array. This may not make much sense for integer stacks, but stack items can be large for example stacks of employees, students, etc where every item is of hundreds of bytes. For such large stacks, the extra space used is comparatively very less as we use two *integer* arrays as extra space.

Following are the two extra arrays are used:

1) *top[]*: This is of size k and stores indexes of top elements in all stacks.

2) *next[]*: This is of size n and stores indexes of next item for the items in array `arr[]`. Here

`arr[]` is actual array that stores k stacks.

Together with k stacks, a stack of free slots in `arr[]` is also maintained. The top of this stack is stored in a variable 'free'.

All entries in `top[]` are initialized as -1 to indicate that all stacks are empty. All entries `next[i]` are initialized as $i+1$ because all slots are free initially and pointing to next slot. Top of free stack, 'free' is initialized as 0.

Following is implementation of the above idea.

C++

```
// A C++ program to demonstrate implementation of k stacks in a single
// array in time and space efficient way
#include<iostream>
#include<climits>
using namespace std;

// A C++ class to represent k stacks in a single array of size n
class kStacks
{
    int *arr;    // Array of size n to store actual content to be stored in stacks
    int *top;    // Array of size k to store indexes of top elements of stacks
    int *next;   // Array of size n to store next entry in all stacks
                // and free list

    int n, k;
    int free;   // To store beginning index of free list
public:
    //constructor to create k stacks in an array of size n
    kStacks(int k, int n);

    // A utility function to check if there is space available
    bool isFull() { return (free == -1); }

    // To push an item in stack number 'sn' where sn is from 0 to k-1
    void push(int item, int sn);

    // To pop an from stack number 'sn' where sn is from 0 to k-1
    int pop(int sn);

    // To check whether stack number 'sn' is empty or not
    bool isEmpty(int sn) { return (top[sn] == -1); }
};

//constructor to create k stacks in an array of size n
kStacks::kStacks(int k1, int n1)
{
    // Initialize n and k, and allocate memory for all arrays
    k = k1, n = n1;
```

```
arr = new int[n];
top = new int[k];
next = new int[n];

// Initialize all stacks as empty
for (int i = 0; i < k; i++)
    top[i] = -1;

// Initialize all spaces as free
free = 0;
for (int i=0; i<n-1; i++)
    next[i] = i+1;
next[n-1] = -1; // -1 is used to indicate end of free list
}

// To push an item in stack number 'sn' where sn is from 0 to k-1
void kStacks::push(int item, int sn)
{
    // Overflow check
    if (isFull())
    {
        cout << "\nStack Overflow\n";
        return;
    }

    int i = free;        // Store index of first free slot

    // Update index of free slot to index of next slot in free list
    free = next[i];

    // Update next of top and then top for stack number 'sn'
    next[i] = top[sn];
    top[sn] = i;

    // Put the item in array
    arr[i] = item;
}

// To pop an from stack number 'sn' where sn is from 0 to k-1
int kStacks::pop(int sn)
{
    // Underflow check
    if (isEmpty(sn))
    {
        cout << "\nStack Underflow\n";
        return INT_MAX;
    }
}
```

```
// Find index of top item in stack number 'sn'
int i = top[sn];

top[sn] = next[i]; // Change top to store next of previous top

// Attach the previous top to the beginning of free list
next[i] = free;
free = i;

// Return the previous top item
return arr[i];
}

/* Driver program to test twStacks class */
int main()
{
    // Let us create 3 stacks in an array of size 10
    int k = 3, n = 10;
    kStacks ks(k, n);

    // Let us put some items in stack number 2
    ks.push(15, 2);
    ks.push(45, 2);

    // Let us put some items in stack number 1
    ks.push(17, 1);
    ks.push(49, 1);
    ks.push(39, 1);

    // Let us put some items in stack number 0
    ks.push(11, 0);
    ks.push(9, 0);
    ks.push(7, 0);

    cout << "Popped element from stack 2 is " << ks.pop(2) << endl;
    cout << "Popped element from stack 1 is " << ks.pop(1) << endl;
    cout << "Popped element from stack 0 is " << ks.pop(0) << endl;

    return 0;
}
```

Java

```
// Java program to demonstrate implementation of k stacks in a single
// array in time and space efficient way

public class GFG
```

```
{
    // A Java class to represent k stacks in a single array of size n
    static class KStack
    {
        int arr[];    // Array of size n to store actual content to be stored in stacks
        int top[];    // Array of size k to store indexes of top elements of stacks
        int next[];   // Array of size n to store next entry in all stacks
                        // and free list

        int n, k;
        int free; // To store beginning index of free list

        //constructor to create k stacks in an array of size n
        KStack(int k1, int n1)
        {
            // Initialize n and k, and allocate memory for all arrays
            k = k1;
            n = n1;
            arr = new int[n];
            top = new int[k];
            next = new int[n];

            // Initialize all stacks as empty
            for (int i = 0; i < k; i++)
                top[i] = -1;

            // Initialize all spaces as free
            free = 0;
            for (int i = 0; i < n - 1; i++)
                next[i] = i + 1;
            next[n - 1] = -1; // -1 is used to indicate end of free list
        }

        // A utility function to check if there is space available
        boolean isFull()
        {
            return (free == -1);
        }

        // To push an item in stack number 'sn' where sn is from 0 to k-1
        void push(int item, int sn)
        {
            // Overflow check
            if (isFull())
            {
                System.out.println("Stack Overflow");
                return;
            }
        }
    }
}
```

```
int i = free; // Store index of first free slot

// Update index of free slot to index of next slot in free list
free = next[i];

// Update next of top and then top for stack number 'sn'
next[i] = top[sn];
top[sn] = i;

// Put the item in array
arr[i] = item;
}

// To pop an from stack number 'sn' where sn is from 0 to k-1
int pop(int sn)
{
    // Underflow check
    if (isEmpty(sn))
    {
        System.out.println("Stack Underflow");
        return Integer.MAX_VALUE;
    }

    // Find index of top item in stack number 'sn'
    int i = top[sn];

    top[sn] = next[i]; // Change top to store next of previous top

    // Attach the previous top to the beginning of free list
    next[i] = free;
    free = i;

    // Return the previous top item
    return arr[i];
}

// To check whether stack number 'sn' is empty or not
boolean isEmpty(int sn)
{
    return (top[sn] == -1);
}

}

// Driver program
public static void main(String[] args)
{
    // Let us create 3 stacks in an array of size 10
```

```
int k = 3, n = 10;

KStack ks = new KStack(k, n);

ks.push(15, 2);
ks.push(45, 2);

// Let us put some items in stack number 1
ks.push(17, 1);
ks.push(49, 1);
ks.push(39, 1);

// Let us put some items in stack number 0
ks.push(11, 0);
ks.push(9, 0);
ks.push(7, 0);

System.out.println("Popped element from stack 2 is " + ks.pop(2));
System.out.println("Popped element from stack 1 is " + ks.pop(1));
System.out.println("Popped element from stack 0 is " + ks.pop(0));
}
}
```

//This code is Contributed by Sumit Ghosh

Output:

```
Popped element from stack 2 is 45
Popped element from stack 1 is 39
Popped element from stack 0 is 7
```

Time complexities of operations push() and pop() is $O(1)$.

The best part of above implementation is, if there is a slot available in stack, then an item can be pushed in any of the stacks, i.e., no wastage of space.

This article is contributed by **Sachin**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/efficiently-implement-k-stacks-single-array/>

Chapter 46

How to implement stack using priority queue or heap?

How to implement stack using priority queue or heap? - GeeksforGeeks

How to Implement stack using a priority queue(using min heap)?.

Asked In: Microsoft, Adobe.

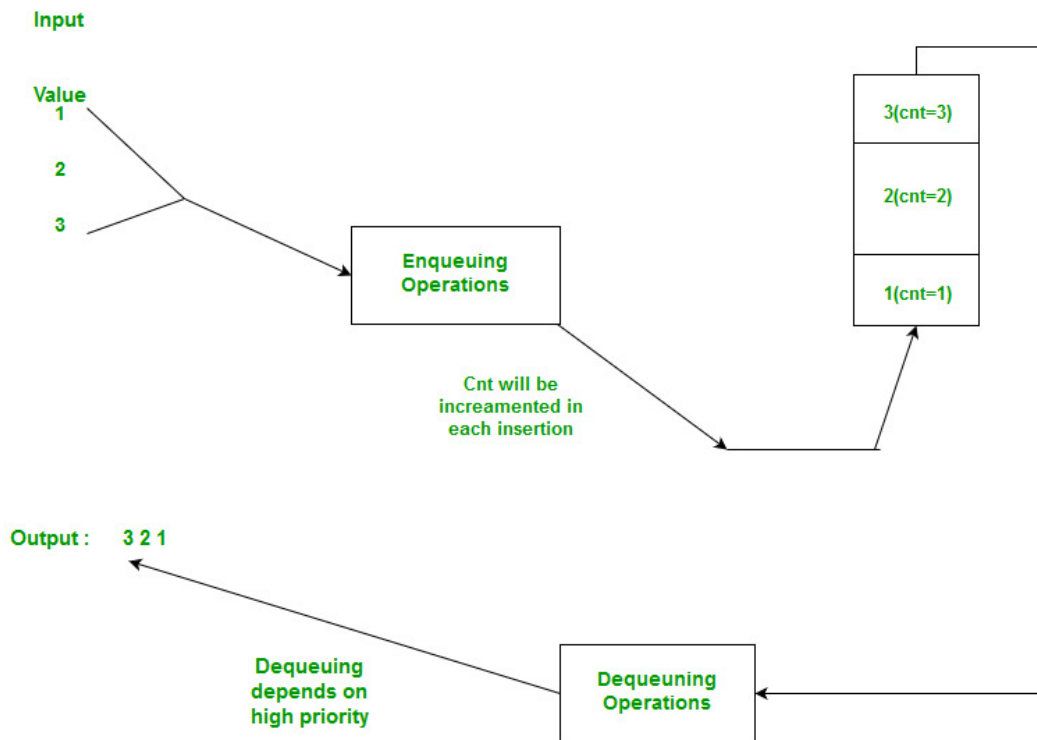
Solution:

In priority queue, we assign priority to the elements that are being pushed. A stack requires elements to be processed in Last in First Out manner. The idea is to associate a count that determines when it was pushed. This count works as a key for the priority queue.

So the implementation of stack uses a priority queue of pairs, with the first element serving as the key.

```
pair <int, int> (key, value)
```

See Below Image to understand Better



Below is C++ implementation of the idea.

```
// C++ program to implement a stack using
// Priority queue(min heap)
#include<bits/stdc++.h>
using namespace std;

typedef pair<int, int> pi;

// User defined stack class
class Stack{

    // cnt is used to keep track of the number of
    //elements in the stack and also serves as key
    //for the priority queue.
    int cnt;
    priority_queue<pair<int, int> > pq;
public:
    Stack():cnt(0){}
    void push(int n);
    void pop();
    int top();
    bool isEmpty();
};
```



```
// push function increases cnt by 1 and
// inserts this cnt with the original value.
void Stack::push(int n){
    cnt++;
    pq.push(pi(cnt, n));
}

// pops element and reduces count.
void Stack::pop(){
    if(pq.empty()){ cout<<"Nothing to pop!!!";}
    cnt--;
    pq.pop();
}

// returns the top element in the stack using
// cnt as key to determine top(highest priority),
// default comparator for pairs works fine in this case
int Stack::top(){
    pi temp=pq.top();
    return temp.second;
}

// return true if stack is empty
bool Stack::isEmpty(){
    return pq.empty();
}

// Driver code
int main()
{
    Stack* s=new Stack();
    s->push(1);
    s->push(2);
    s->push(3);
    while(!s->isEmpty()){
        cout<<s->top()<<endl;
        s->pop();
    }
}
```

Output:

```
3
2
1
```

Now, as we can see this implementation takes $O(\log n)$ time for both push and pop operations. This can be slightly optimized by using fibonacci heap implementation of priority queue which would give us $O(1)$ time complexity for push operation, but pop still requires $O(\log n)$ time.

Source

<https://www.geeksforgeeks.org/implement-stack-using-priority-queue-or-heap/>

Chapter 47

Identify and mark unmatched parenthesis in an expression

Identify and mark unmatched parenthesis in an expression - GeeksforGeeks

Given an expression, find and mark matched and unmatched parenthesis in it. We need to replace all balanced opening parenthesis with 0, balanced closing parenthesis with 1 and all unbalanced with -1.

Examples:

Input : ((a)
Output : -10a1

Input : (a))
Output : 0a1-1

Input : (((abc))((d))))
Output : 000abc1100d111-1-1

The idea is based on [stack](#). We run a loop from the start of the string upto end and for every '(', we push it into stack. If stack is empty and we encounter an closing bracket ')' the we replace -1 at that index of the strig. Else we replace all opening brackets '(' with 0 and closing brackets with 1. Then pop from the stack.

```
// CPP program to mark balanced and unbalanced
// parenthesis.
#include <bits/stdc++.h>
using namespace std;

void identifyParenthesis(string a)
{
```

```
stack<int> st;

// run the loop upto end of the string
for (int i = 0; i < a.length(); i++) {

    // if a[i] is opening bracket then push
    // into stack
    if (a[i] == '(')
        st.push(i);

    // if a[i] is closing bracket ')'
    else if (a[i] == ')') {

        // If this closing bracket is unmatched
        if (st.empty())
            a.replace(i, 1, "-1");

        else {

            // replace all opening brackets with 0
            // and closing brackets with 1
            a.replace(i, 1, "1");
            a.replace(st.top(), 1, "0");
            st.pop();
        }
    }
}

// if stack is not empty then pop out all
// elements from it and replace -1 at that
// index of the string
while (!st.empty()) {
    a.replace(st.top(), 1, "-1");
    st.pop();
}

// print final string
cout << a << endl;
}

// Driver code
int main()
{
    string str = "(a)";
    identifyParenthesis(str);
    return 0;
}
```

Output:

0a1-1

Improved By : [HARDY_123](#)

Source

<https://www.geeksforgeeks.org/identify-mark-unmatched-parenthesis-expression/>

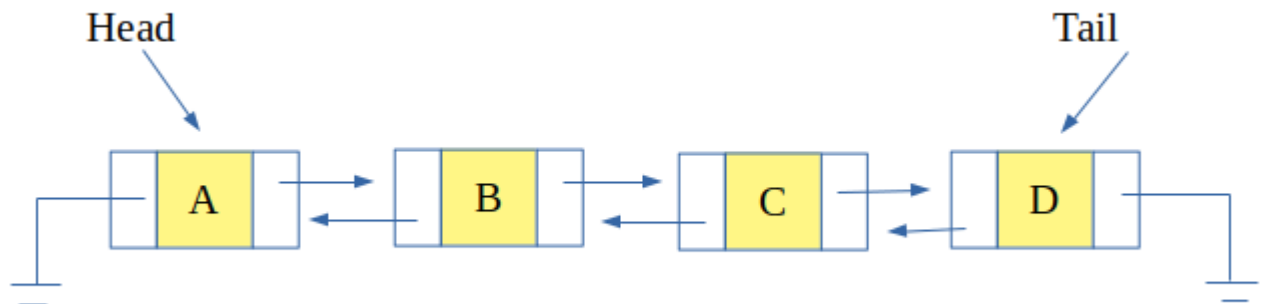
Chapter 48

Implement Stack and Queue using Deque

Implement Stack and Queue using Deque - GeeksforGeeks

Deque also known as **double ended queue**, as name suggests is a special kind of queue in which insertions and deletions can be done at the last as well as at the beginning.

A link-list representation of deque is such that each node points to the next node as well as the previous node. So that insertion and deletions take constant time at both the beginning and the last.



Now, deque can be used to implement a stack and queue. One simply needs to understand how deque can be made to work as a stack or a queue.

The functions of deque to tweak them to work as stack and queue are listed below.

DEQUE	STACK	QUEUE
size()	size()	size()
isEmpty()	isEmpty()	isEmpty()
Insert_First()	-	-
Insert_Last()	Push()	Enqueue()
Remove_First()	-	Dequeue()
Remove_Last()	Pop()	-

Examples: Stack

Input : Stack : 1 2 3
 Push(4)
 Output : Stack : 1 2 3 4

Input : Stack : 1 2 3
 Pop()
 Output : Stack : 1 2

Examples: Queue

Input: Queue : 1 2 3
 Enqueue(4)
 Output: Queue : 1 2 3 4

Input: Queue : 1 2 3
 Dequeue()
 Output: Queue : 2 3

```
// CPP Program to implement stack and queue using Deque
#include <iostream>
using namespace std;

// structure for a node of deque
struct DQueueNode {
    int value;
    DQueueNode* next;
    DQueueNode* prev;
};
```

```
// Implementation of deque class
class deque {
private:

    // pointers to head and tail of deque
    DQueNode* head;
    DQueNode* tail;

public:
    // constructor
    deque()
    {
        head = tail = NULL;
    }

    // if list is empty
    bool isEmpty()
    {
        if (head == NULL)
            return true;
        return false;
    }

    // count the number of nodes in list
    int size()
    {
        // if list is not empty
        if (!isEmpty()) {
            DQueNode* temp = head;
            int len = 0;
            while (temp != NULL) {
                len++;
                temp = temp->next;
            }
            return len;
        }
        return 0;
    }

    // insert at the first position
    void insert_first(int element)
    {
        // allocating node of DQueNode type
        DQueNode* temp = new DQueNode[sizeof(DQueNode)];
        temp->value = element;

        // if the element is first element
```



```
        if (head == NULL) {
            head = tail = temp;
            temp->next = temp->prev = NULL;
        }
        else {
            head->prev = temp;
            temp->next = head;
            temp->prev = NULL;
            head = temp;
        }
    }

    // insert at last position of deque
    void insert_last(int element)
    {
        // allocating node of DQueNode type
        DQueNode* temp = new DQueNode[sizeof(DQueNode)];
        temp->value = element;

        // if element is the first element
        if (head == NULL) {
            head = tail = temp;
            temp->next = temp->prev = NULL;
        }
        else {
            tail->next = temp;
            temp->next = NULL;
            temp->prev = tail;
            tail = temp;
        }
    }

    // remove element at the first position
    void remove_first()
    {
        // if list is not empty
        if (!isEmpty()) {
            DQueNode* temp = head;
            head = head->next;
            head->prev = NULL;
            free(temp);
            return;
        }
        cout << "List is Empty" << endl;
    }

    // remove element at the last position
    void remove_last()
```

```
{
    // if list is not empty
    if (!isEmpty()) {
        DQueNode* temp = tail;
        tail = tail->prev;
        tail->next = NULL;
        free(temp);
        return;
    }
    cout << "List is Empty" << endl;
}

// displays the elements in deque
void display()
{
    // if list is not empty
    if (!isEmpty()) {
        DQueNode* temp = head;
        while (temp != NULL) {
            cout << temp->value << " ";
            temp = temp->next;
        }
        cout << endl;
        return;
    }
    cout << "List is Empty" << endl;
}

};

// Class to implement stack using Deque
class Stack : public deque {
public:
    // push to push element at top of stack
    // using insert at last function of deque
    void push(int element)
    {
        insert_last(element);
    }

    // pop to remove element at top of stack
    // using remove at last function of deque
    void pop()
    {
        remove_last();
    }
};

// class to implement queue using deque
```

```
class Queue : public deque {
public:
    // enqueue to insert element at last
    // using insert at last function of deque
    void enqueue(int element)
    {
        insert_last(element);
    }

    // dequeue to remove element from first
    // using remove at first function of deque
    void dequeue()
    {
        remove_first();
    }
};

// Driver Code
int main()
{
    // object of Stack
    Stack stk;

    // push 7 and 8 at top of stack
    stk.push(7);
    stk.push(8);
    cout << "Stack: ";
    stk.display();

    // pop an element
    stk.pop();
    cout << "Stack: ";
    stk.display();

    // object of Queue
    Queue que;

    // insert 12 and 13 in queue
    que.enqueue(12);
    que.enqueue(13);
    cout << "Queue: ";
    que.display();

    // delete an element from queue
    que.dequeue();
    cout << "Queue: ";
    que.display();
}
```

```
    cout << "Size of Stack is " << stk.size() << endl;
    cout << "Size of Queue is " << que.size() << endl;
    return 0;
}
```

Output:

```
Stack: 7 8
Stack: 7
Queue: 12 13
Queue: 13
Size of Stack is 1
Size of Queue is 1
```

Source

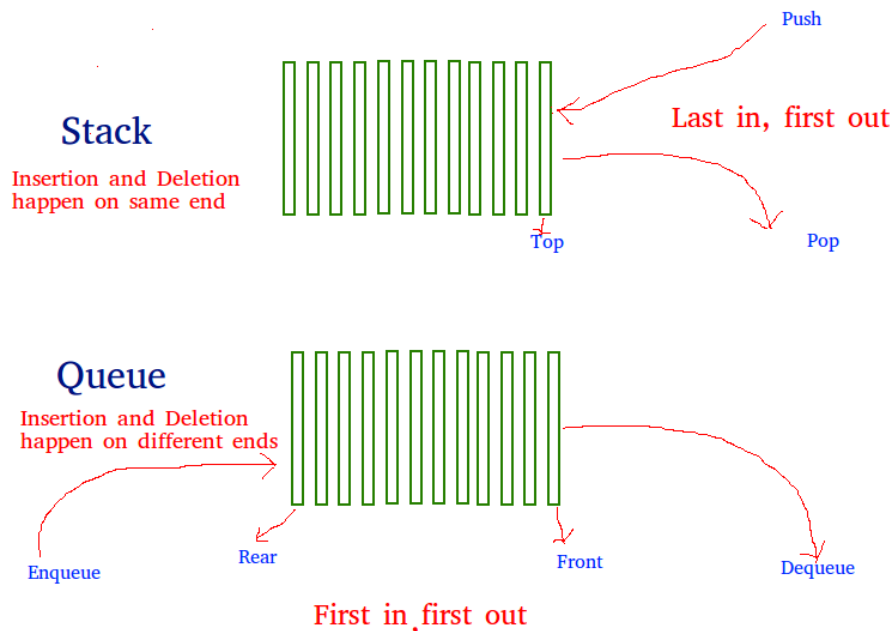
<https://www.geeksforgeeks.org/implement-stack-queue-using-deque/>

Chapter 49

Implement Stack using Queues

Implement Stack using Queues - GeeksforGeeks

The problem is opposite of [this](#) post. We are given a Queue data structure that supports standard operations like enqueue() and dequeue(). We need to implement a Stack data structure using only instances of Queue and queue operations allowed on the instances.



A stack can be implemented using two queues. Let stack to be implemented be 's' and queues used to implement be 'q1' and 'q2'. Stack 's' can be implemented in two ways:

Method 1 (By making push operation costly)

This method makes sure that newly entered element is always at the front of 'q1', so that pop operation just dequeues from 'q1'. 'q2' is used to put every new element at front of 'q1'.

```
push(s, x) // x is the element to be pushed and s is stack
    1) Enqueue x to q2
    2) One by one dequeue everything from q1 and enqueue to q2.
    3) Swap the names of q1 and q2
// Swapping of names is done to avoid one more movement of all elements
// from q2 to q1.

pop(s)
    1) Dequeue an item from q1 and return it.

/* Program to implement a stack using
two queue */
#include<bits/stdc++.h>
using namespace std;

class Stack
{
    // Two inbuilt queues
    queue<int> q1, q2;

    // To maintain current number of
    // elements
    int curr_size;

public:
    Stack()
    {
        curr_size = 0;
    }

    void push(int x)
    {
        curr_size++;

        // Push x first in empty q2
        q2.push(x);

        // Push all the remaining
        // elements in q1 to q2.
        while (!q1.empty())
        {
            q2.push(q1.front());
            q1.pop();
        }

        // swap the names of two queues
        queue<int> q = q1;
```

```
        q1 = q2;
        q2 = q;
    }

    void pop(){

        // if no elements are there in q1
        if (q1.empty())
            return ;
        q1.pop();
        curr_size--;
    }

    int top()
    {
        if (q1.empty())
            return -1;
        return q1.front();
    }

    int size()
    {
        return curr_size;
    }
};

// driver code
int main()
{
    Stack s;
    s.push(1);
    s.push(2);
    s.push(3);

    cout << "current size: " << s.size()
          << endl;
    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;

    cout << "current size: " << s.size()
          << endl;
    return 0;
}
// This code is contributed by Chhavi
```

Output :

```
current size: 3
3
2
1
current size: 1
```

Method 2 (By making pop operation costly)

In push operation, the new element is always enqueued to q1. In pop() operation, if q2 is empty then all the elements except the last, are moved to q2. Finally the last element is dequeued from q1 and returned.

```
push(s, x)
    1) Enqueue x to q1 (assuming size of q1 is unlimited).

pop(s)
    1) One by one dequeue everything except the last element from q1 and enqueue to q2.
    2) Dequeue the last item of q1, the dequeued item is result, store it.
    3) Swap the names of q1 and q2
    4) Return the item stored in step 2.
// Swapping of names is done to avoid one more movement of all elements
// from q2 to q1.

/* Program to implement a stack
using two queue */
#include<bits/stdc++.h>
using namespace std;

class Stack
{
    queue<int> q1, q2;
    int curr_size;

public:
    Stack()
    {
        curr_size = 0;
    }

    void pop()
    {
        if (q1.empty())
            return;

        // Leave one element in q1 and
```



```
// push others in q2.
while (q1.size() != 1)
{
    q2.push(q1.front());
    q1.pop();
}

// Pop the only left element
// from q1
q1.pop();
curr_size--;

// swap the names of two queues
queue<int> q = q1;
q1 = q2;
q2 = q;
}

void push(int x)
{
    q1.push(x);
    curr_size++;
}

int top()
{
    if (q1.empty())
        return -1;

    while( q1.size() != 1 )
    {
        q2.push(q1.front());
        q1.pop();
    }

    // last pushed element
    int temp = q1.front();

    // to empty the auxiliary queue after
    // last operation
    q1.pop();

    // push last element to q2
    q2.push(temp);

    // swap the two queues names
    queue<int> q = q1;
    q1 = q2;
```

```
        q2 = q;
        return temp;
    }

    int size()
    {
        return curr_size;
    }
};

// driver code
int main()
{
    Stack s;
    s.push(1);
    s.push(2);
    s.push(3);
    s.push(4);

    cout << "current size: " << s.size()
          << endl;
    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;
    cout << "current size: " << s.size()
          << endl;
    return 0;
}
// This code is contributed by Chhavi
```

Output :

```
current size: 4
4
3
2
current size: 2
```

References:

[Implement Stack using Two Queues](#)

This article is compiled by **Sumit Jain** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/implement-stack-using-queue/>

Chapter 50

Implement a stack using single queue

Implement a stack using single queue - GeeksforGeeks

We are given queue data structure, the task is to implement stack using only given queue data structure.

We have discussed [a solution that uses two queues](#). In this article, a new solution is discussed that uses only one queue. This solution assumes that we can find size of queue at any point. The idea is to keep newly inserted element always at front, keeping order of previous elements same. Below are complete steps.

```
// x is the element to be pushed and s is stack
push(s, x)
    1) Let size of q be s.
    1) Enqueue x to q
    2) One by one Dequeue s items from queue and enqueue them.

// Removes an item from stack
pop(s)
    1) Dequeue an item from q
```

Below is implementation of the idea.

C++

```
// C++ program to implement a stack using
// single queue
#include<bits/stdc++.h>
using namespace std;
```

```
// User defined stack that uses a queue
class Stack
{
    queue<int>q;
public:
    void push(int val);
    void pop();
    int top();
    bool empty();
};

// Push operation
void Stack::push(int val)
{
    // Get previous size of queue
    int s = q.size();

    // Push current element
    q.push(val);

    // Pop (or Dequeue) all previous
    // elements and put them after current
    // element
    for (int i=0; i<s; i++)
    {
        // this will add front element into
        // rear of queue
        q.push(q.front());

        // this will delete front element
        q.pop();
    }
}

// Removes the top element
void Stack::pop()
{
    if (q.empty())
        cout << "No elements\n";
    else
        q.pop();
}

// Returns top of stack
int Stack::top()
{
    return (q.empty())? -1 : q.front();
}
```

```
}

// Returns true if Stack is empty else false
bool Stack::empty()
{
    return (q.empty());
}

// Driver code
int main()
{
    Stack s;
    s.push(10);
    s.push(20);
    cout << s.top() << endl;
    s.pop();
    s.push(30);
    s.pop();
    cout << s.top() << endl;
    return 0;
}
```

Java

```
// Java program to implement stack using a
// single queue

import java.util.LinkedList;
import java.util.Queue;

public class stack
{
    Queue<Integer> q = new LinkedList<Integer>();

    // Push operation
    void push(int val)
    {
        // get previous size of queue
        int size = q.size();

        // Add current element
        q.add(val);

        // Pop (or Dequeue) all previous
        // elements and put them after current
        // element
        for (int i = 0; i < size; i++)
        {
```

```
        // this will add front element into
        // rear of queue
        int x = q.remove();
        q.add(x);
    }
}

// Removes the top element
int pop()
{
    if (q.isEmpty())
    {
        System.out.println("No elements");
        return -1;
    }
    int x = q.remove();
    return x;
}

// Returns top of stack
int top()
{
    if (q.isEmpty())
        return -1;
    return q.peek();
}

// Returns true if Stack is empty else false
boolean isEmpty()
{
    return q.isEmpty();
}

// Driver program to test above methods
public static void main(String[] args)
{
    stack s = new stack();
    s.push(10);
    s.push(20);
    System.out.println("Top element : " + s.top());
    s.pop();
    s.push(30);
    s.pop();
    System.out.println("Top element : " + s.top());
}

// This code is contributed by Rishabh Mahrsee
```

Output :

20
10

Source

<https://www.geeksforgeeks.org/implement-a-stack-using-single-queue/>

Chapter 51

Implement two stacks in an array

Implement two stacks in an array - GeeksforGeeks

Create a data structure *twoStacks* that represents two stacks. Implementation of *twoStacks* should use only one array, i.e., both stacks should use the same array for storing elements. Following functions must be supported by *twoStacks*.

push1(int x) -> pushes x to first stack

push2(int x) -> pushes x to second stack

pop1() -> pops an element from first stack and return the popped element

pop2() -> pops an element from second stack and return the popped element

Implementation of *twoStack* should be space efficient.

Method 1 (Divide the space in two halves)

A simple way to implement two stacks is to divide the array in two halves and assign the half half space to two stacks, i.e., use arr[0] to arr[n/2] for stack1, and arr[(n/2) + 1] to arr[n-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be n.

The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in arr[]. For example, say the array size is 6 and we push 3 elements to stack1 and do not push anything to second stack2. When we push 4th element to stack1, there will be overflow even if we have space for 3 more elements in array.

Method 2 (A space efficient implementation)

This method efficiently utilizes the available space. It doesn't cause an overflow if there is space available in arr[]. The idea is to start two stacks from two extreme corners of arr[]. stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0. The stack2 starts from the rightmost corner, the first element in stack2 is pushed at index (n-1). Both stacks grow (or shrink) in opposite direction. To check for overflow, all we need to check

is for space between top elements of both stacks. This check is highlighted in the below code.

C++

```
#include<iostream>
#include<stdlib.h>

using namespace std;

class twoStacks
{
    int *arr;
    int size;
    int top1, top2;
public:
    twoStacks(int n) // constructor
    {
        size = n;
        arr = new int[n];
        top1 = -1;
        top2 = size;
    }

    // Method to push an element x to stack1
    void push1(int x)
    {
        // There is at least one empty space for new element
        if (top1 < top2 - 1)
        {
            top1++;
            arr[top1] = x;
        }
        else
        {
            cout << "Stack Overflow";
            exit(1);
        }
    }

    // Method to push an element x to stack2
    void push2(int x)
    {
        // There is at least one empty space for new element
        if (top1 < top2 - 1)
        {
            top2--;
            arr[top2] = x;
        }
    }
}
```

```
        else
        {
            cout << "Stack Overflow";
            exit(1);
        }
    }

    // Method to pop an element from first stack
    int pop1()
    {
        if (top1 >= 0 )
        {
            int x = arr[top1];
            top1--;
            return x;
        }
        else
        {
            cout << "Stack UnderFlow";
            exit(1);
        }
    }

    // Method to pop an element from second stack
    int pop2()
    {
        if (top2 < size)
        {
            int x = arr[top2];
            top2++;
            return x;
        }
        else
        {
            cout << "Stack UnderFlow";
            exit(1);
        }
    }
}

};

/* Driver program to test twStacks class */
int main()
{
    twoStacks ts(5);
    ts.push1(5);
    ts.push2(10);
    ts.push2(15);
```

```
ts.push1(11);
ts.push2(7);
cout << "Popped element from stack1 is " << ts.pop1();
ts.push2(40);
cout << "\nPopped element from stack2 is " << ts.pop2();
return 0;
}
```

Java

```
// Java program to implement two stacks in a
// single array
class TwoStacks
{
    int size;
    int top1, top2;
    int arr[];

    // Constructor
    TwoStacks(int n)
    {
        arr = new int[n];
        size = n;
        top1 = -1;
        top2 = size;
    }

    // Method to push an element x to stack1
    void push1(int x)
    {
        // There is at least one empty space for
        // new element
        if (top1 < top2 - 1)
        {
            top1++;
            arr[top1] = x;
        }
        else
        {
            System.out.println("Stack Overflow");
            System.exit(1);
        }
    }

    // Method to push an element x to stack2
    void push2(int x)
    {
        // There is at least one empty space for
```

```
// new element
if (top1 < top2 -1)
{
    top2--;
    arr[top2] = x;
}
else
{
    System.out.println("Stack Overflow");
    System.exit(1);
}
}

// Method to pop an element from first stack
int pop1()
{
    if (top1 >= 0)
    {
        int x = arr[top1];
        top1--;
        return x;
    }
    else
    {
        System.out.println("Stack Underflow");
        System.exit(1);
    }
    return 0;
}

// Method to pop an element from second stack
int pop2()
{
    if(top2 < size)
    {
        int x =arr[top2];
        top2++;
        return x;
    }
    else
    {
        System.out.println("Stack Underflow");
        System.exit(1);
    }
    return 0;
}
```

```
// Driver program to test twoStack class
public static void main(String args[])
{
    TwoStacks ts = new TwoStacks(5);
    ts.push1(5);
    ts.push2(10);
    ts.push2(15);
    ts.push1(11);
    ts.push2(7);
    System.out.println("Popped element from" +
        " stack1 is " + ts.pop1());
    ts.push2(40);
    System.out.println("Popped element from" +
        " stack2 is " + ts.pop2());
}
}
// This code has been contributed by
// Amit Khandelwal(Amit Khandelwal 1).
```

Python

```
# Python Script to Implement two stacks in a list
class twoStacks:

    def __init__(self, n):      #constructor
        self.size = n
        self.arr = [None] * n
        self.top1 = -1
        self.top2 = self.size

    # Method to push an element x to stack1
    def push1(self, x):

        # There is at least one empty space for new element
        if self.top1 < self.top2 - 1 :
            self.top1 = self.top1 + 1
            self.arr[self.top1] = x

        else:
            print("Stack Overflow ")
            exit(1)

    # Method to push an element x to stack2
    def push2(self, x):

        # There is at least one empty space for new element
        if self.top1 < self.top2 - 1:
            self.top2 = self.top2 - 1
```

```
        self.arr[self.top2] = x

    else :
        print("Stack Overflow ")
        exit(1)

# Method to pop an element from first stack
def pop1(self):
    if self.top1 >= 0:
        x = self.arr[self.top1]
        self.top1 = self.top1 -1
        return x
    else:
        print("Stack Underflow ")
        exit(1)

# Method to pop an element from second stack
def pop2(self):
    if self.top2 < self.size:
        x = self.arr[self.top2]
        self.top2 = self.top2 + 1
        return x
    else:
        print("Stack Underflow ")
        exit()

# Driver program to test twoStacks class
ts = twoStacks(5)
ts.push1(5)
ts.push2(10)
ts.push2(15)
ts.push1(11)
ts.push2(7)

print("Popped element from stack1 is " + str(ts.pop1()))
ts.push2(40)
print("Popped element from stack2 is " + str(ts.pop2()))

# This code is contributed by Sunny Karira
```

Output:

```
Popped element from stack1 is 11
Popped element from stack2 is 40
```

Time complexity of all 4 operations of *twoStack* is $O(1)$.

We will extend to 3 stacks in an array in a separate post.

Improved By : [siddhartha33](#)

Source

<https://www.geeksforgeeks.org/implement-two-stacks-in-an-array/>

Chapter 52

Infix to Prefix conversion using two stacks

Infix to Prefix conversion using two stacks - GeeksforGeeks

Infix : An expression is called the Infix expression if the operator appears in between the operands in the expression. Simply of the form (operand1 operator operand2).

Example : $(A+B) * (C-D)$

Prefix : An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2).

Example : $*+AB-CD$ (Infix : $(A+B) * (C-D)$)

Given an Infix expression, convert it into a Prefix expression using two stacks.

Examples:

Input : $A * B + C / D$

Output : $+ * A B / C D$

Input : $(A - B/C) * (A/K-L)$

Output : $*-A/BC-/AKL$

The idea is to use one stack for storing operators and other to store operands. The stepwise algo is:

1. Traverse the infix expression and check if given character is an operator or an operand.
2. If it is an operand, then push it into operand stack.

3. If it is an operator, then check if priority of current operator is greater than or less than or equal to the operator at top of the stack. If priority is greater, then push operator into operator stack. Otherwise pop two operands from operand stack, pop operator from operator stack and push string **operator + operand1 + operand 2** into operand stack. Keep popping from both stacks and pushing result into operand stack until priority of current operator is less than or equal to operator at top of the operator stack.
4. If current character is '(', then push it into operator stack.
5. If current character is ')', then check if top of operator stack is opening bracket or not. If not pop two operands from operand stack, pop operator from operator stack and push string **operator + operand1 + operand 2** into operand stack. Keep popping from both stacks and pushing result into operand stack until top of operator stack is an opening bracket.
6. The final prefix expression is present at top of operand stack.

Below is the implementation of above algorithm:

C++

```
// CPP program to convert infix to prefix.
#include <bits/stdc++.h>
using namespace std;

// Function to check if given character is
// an operator or not.
bool isOperator(char c)
{
    return (!isalpha(c) && !isdigit(c));
}

// Function to find priority of given
// operator.
int getPriority(char C)
{
    if (C == '-' || C == '+')
        return 1;
    else if (C == '*' || C == '/')
        return 2;
    else if (C == '^')
        return 3;
    return 0;
}

// Function that converts infix
// expression to prefix expression.
string infixToPrefix(string infix)
{

```

```
// stack for operators.
stack<char> operators;

// stack for operands.
stack<string> operands;

for (int i = 0; i < infix.length(); i++) {

    // If current character is an
    // opening bracket, then
    // push into the operators stack.
    if (infix[i] == '(') {
        operators.push(infix[i]);
    }

    // If current character is a
    // closing bracket, then pop from
    // both stacks and push result
    // in operands stack until
    // matching opening bracket is
    // not found.
    else if (infix[i] == ')') {
        while (!operators.empty() &&
            operators.top() != '(') {

            // operand 1
            string op1 = operands.top();
            operands.pop();

            // operand 2
            string op2 = operands.top();
            operands.pop();

            // operator
            char op = operators.top();
            operators.pop();

            // Add operands and operator
            // in form operator +
            // operand1 + operand2.
            string tmp = op + op2 + op1;
            operands.push(tmp);
        }

        // Pop opening bracket from
        // stack.
        operators.pop();
    }
}
```

```
// If current character is an
// operand then push it into
// operands stack.
else if (!isOperator(infix[i])) {
    operands.push(string(1, infix[i]));
}

// If current character is an
// operator, then push it into
// operators stack after popping
// high priority operators from
// operators stack and pushing
// result in operands stack.
else {
    while (!operators.empty() &&
        getPriority(infix[i]) <=
            getPriority(operators.top())) {

        string op1 = operands.top();
        operands.pop();

        string op2 = operands.top();
        operands.pop();

        char op = operators.top();
        operators.pop();

        string tmp = op + op2 + op1;
        operands.push(tmp);
    }

    operators.push(infix[i]);
}

// Pop operators from operators stack
// until it is empty and add result
// of each pop operation in
// operands stack.
while (!operators.empty()) {
    string op1 = operands.top();
    operands.pop();

    string op2 = operands.top();
    operands.pop();

    char op = operators.top();
```

```
        operators.pop();

        string tmp = op + op2 + op1;
        operands.push(tmp);
    }

    // Final prefix expression is
    // present in operands stack.
    return operands.top();
}

// Driver code
int main()
{
    string s = "(A-B/C)*(A/K-L)";
    cout << infixToPrefix(s);
    return 0;
}
```

Java

```
// Java program to convert
// infix to prefix.
import java.util.*;
class GFG
{
    // Function to check if
    // given character is
    // an operator or not.
    static boolean isOperator(char c)
    {
        return (!(c >= 'a' && c <= 'z') &&
                !(c >= '0' && c <= '9') &&
                !(c >= 'A' && c <= 'Z'));
    }

    // Function to find priority
    // of given operator.
    static int getPriority(char C)
    {
        if (C == '-' || C == '+')
            return 1;
        else if (C == '*' || C == '/')
            return 2;
        else if (C == '^')
            return 3;
        return 0;
    }
}
```

```
// Function that converts infix
// expression to prefix expression.
static String infixToPrefix(String infix)
{
    // stack for operators.
    Stack<Character> operators = new Stack<Character>();

    // stack for operands.
    Stack<String> operands = new Stack<String>();

    for (int i = 0; i < infix.length(); i++)
    {
        // If current character is an
        // opening bracket, then
        // push into the operators stack.
        if (infix.charAt(i) == '(')
        {
            operators.push(infix.charAt(i));
        }

        // If current character is a
        // closing bracket, then pop from
        // both stacks and push result
        // in operands stack until
        // matching opening bracket is
        // not found.
        else if (infix.charAt(i) == ')')
        {
            while (!operators.empty() &&
                operators.peek() != '(')
            {

                // operand 1
                String op1 = operands.peek();
                operands.pop();

                // operand 2
                String op2 = operands.peek();
                operands.pop();

                // operator
                char op = operators.peek();
                operators.pop();

                // Add operands and operator
                // in form operator +
            }
        }
    }
}
```

```
        // operand1 + operand2.
        String tmp = op + op2 + op1;
        operands.push(tmp);
    }

    // Pop opening bracket
    // from stack.
    operators.pop();
}

// If current character is an
// operand then push it into
// operands stack.
else if (!isOperator(infix.charAt(i)))
{
    operands.push(infix.charAt(i) + "");
}

// If current character is an
// operator, then push it into
// operators stack after popping
// high priority operators from
// operators stack and pushing
// result in operands stack.
else
{
    while (!operators.empty() &&
        getPriority(infix.charAt(i)) <=
        getPriority(operators.peek()))
    {

        String op1 = operands.peek();
        operands.pop();

        String op2 = operands.peek();
        operands.pop();

        char op = operators.peek();
        operators.pop();

        String tmp = op + op2 + op1;
        operands.push(tmp);
    }

    operators.push(infix.charAt(i));
}
}
```

```
// Pop operators from operators
// stack until it is empty and
// operation in add result of
// each pop operands stack.
while (!operators.empty())
{
    String op1 = operands.peek();
    operands.pop();

    String op2 = operands.peek();
    operands.pop();

    char op = operators.peek();
    operators.pop();

    String tmp = op + op2 + op1;
    operands.push(tmp);
}

// Final prefix expression is
// present in operands stack.
return operands.peek();
}

// Driver code
public static void main(String args[])
{
    String s = "(A-B/C)*(A/K-L)";
    System.out.println( infixToPrefix(s));
}
}
```

// This code is contributed
// by Arnab Kundu

Output:

*-A/BC-/AKL

Time Complexity: $O(n)$

Auxiliary Space: $O(n)$

Improved By : [andrew1234](#)

Source

<https://www.geeksforgeeks.org/infix-to-prefix-conversion-using-two-stacks/>

Chapter 53

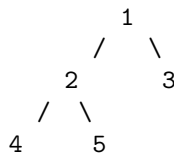
Inorder Tree Traversal without Recursion

Inorder Tree Traversal without Recursion - GeeksforGeeks

Using [Stack](#) is the obvious way to traverse tree without recursion. Below is an algorithm for traversing binary tree using stack. See [this](#) for step wise step execution of the algorithm.

- 1) Create an empty stack S.
- 2) Initialize current node as root
- 3) Push the current node to S and set current = current->left until current is NULL
- 4) If current is NULL and stack is not empty then
 - a) Pop the top item from stack.
 - b) Print the popped item, set current = popped_item->right
 - c) Go to step 3.
- 5) If current is NULL and stack is empty then we are done.

Let us consider the below tree for example



Step 1 Creates an empty stack: S = NULL

Step 2 sets current as address of root: current -> 1

Step 3 Pushes the current node and set current = current->left until current is NULL

```
current -> 1
push 1: Stack S -> 1
current -> 2
push 2: Stack S -> 2, 1
current -> 4
push 4: Stack S -> 4, 2, 1
current = NULL
```

Step 4 pops from S

- a) Pop 4: Stack S -> 2, 1
- b) print "4"
- c) current = NULL /*right of 4 */ and go to step 3

Since current is NULL step 3 doesn't do anything.

Step 4 pops again.

- a) Pop 2: Stack S -> 1
- b) print "2"
- c) current -> 5/*right of 2 */ and go to step 3

Step 3 pushes 5 to stack and makes current NULL

```
Stack S -> 5, 1
current = NULL
```

Step 4 pops from S

- a) Pop 5: Stack S -> 1
- b) print "5"
- c) current = NULL /*right of 5 */ and go to step 3

Since current is NULL step 3 doesn't do anything

Step 4 pops again.

- a) Pop 1: Stack S -> NULL
- b) print "1"
- c) current -> 3 /*right of 5 */

Step 3 pushes 3 to stack and makes current NULL

```
Stack S -> 3
current = NULL
```

Step 4 pops from S

- a) Pop 3: Stack S -> NULL
- b) print "3"
- c) current = NULL /*right of 3 */

Traversal is done now as stack S is empty and current is NULL.

C++

```
// C++ program to print inorder traversal
```

```
// using stack.
#include<bits/stdc++.h>
using namespace std;

/* A binary tree Node has data, pointer to left child
   and a pointer to right child */
struct Node
{
    int data;
    struct Node* left;
    struct Node* right;
    Node (int data)
    {
        this->data = data;
        left = right = NULL;
    }
};

/* Iterative function for inorder tree
   traversal */
void inOrder(struct Node *root)
{
    stack<Node *> s;
    Node *curr = root;

    while (curr != NULL || s.empty() == false)
    {
        /* Reach the left most Node of the
           curr Node */
        while (curr != NULL)
        {
            /* place pointer to a tree node on
               the stack before traversing
               the node's left subtree */
            s.push(curr);
            curr = curr->left;
        }

        /* Current must be NULL at this point */
        curr = s.top();
        s.pop();

        cout << curr->data << " ";

        /* we have visited the node and its
           left subtree. Now, it's right
           subtree's turn */
        curr = curr->right;
    }
}
```

```
    } /* end of while */
}

/* Driver program to test above functions*/
int main()
{
    /* Constructed binary tree is
        1
       / \
      2   3
     / \
    4   5
    */
    struct Node *root = new Node(1);
    root->left      = new Node(2);
    root->right     = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);

    inOrder(root);
    return 0;
}
```

C

```
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* A binary tree tNode has data, pointer to left child
   and a pointer to right child */
struct tNode
{
    int data;
    struct tNode* left;
    struct tNode* right;
};

/* Structure of a stack node. Linked List implementation is used for
   stack. A stack node contains a pointer to tree node and a pointer to
   next stack node */
struct sNode
{
    struct tNode *t;
    struct sNode *next;
};
```

```
/* Stack related functions */
void push(struct sNode** top_ref, struct tNode *t);
struct tNode *pop(struct sNode** top_ref);
bool isEmpty(struct sNode *top);

/* Iterative function for inorder tree traversal */
void inOrder(struct tNode *root)
{
    /* set current to root of binary tree */
    struct tNode *current = root;
    struct sNode *s = NULL; /* Initialize stack s */
    bool done = 0;

    while (!done)
    {
        /* Reach the left most tNode of the current tNode */
        if(current != NULL)
        {
            /* place pointer to a tree node on the stack before traversing
               the node's left subtree */
            push(&s, current);
            current = current->left;
        }

        /* backtrack from the empty subtree and visit the tNode
           at the top of the stack; however, if the stack is empty,
           you are done */
        else
        {
            if (!isEmpty(s))
            {
                current = pop(&s);
                printf("%d ", current->data);

                /* we have visited the node and its left subtree.
                   Now, it's right subtree's turn */
                current = current->right;
            }
            else
                done = 1;
        }
    } /* end of while */
}

/* UTILITY FUNCTIONS */
/* Function to push an item to sNode*/
void push(struct sNode** top_ref, struct tNode *t)
```

```
{
    /* allocate tNode */
    struct sNode* new_tNode =
        (struct sNode*) malloc(sizeof(struct sNode));

    if(new_tNode == NULL)
    {
        printf("Stack Overflow \n");
        getchar();
        exit(0);
    }

    /* put in the data */
    new_tNode->t = t;

    /* link the old list off the new tNode */
    new_tNode->next = (*top_ref);

    /* move the head to point to the new tNode */
    (*top_ref) = new_tNode;
}

/* The function returns true if stack is empty, otherwise false */
bool isEmpty(struct sNode *top)
{
    return (top == NULL)? 1 : 0;
}

/* Function to pop an item from stack*/
struct tNode *pop(struct sNode** top_ref)
{
    struct tNode *res;
    struct sNode *top;

    /*If sNode is empty then error */
    if(isEmpty(*top_ref))
    {
        printf("Stack Underflow \n");
        getchar();
        exit(0);
    }
    else
    {
        top = *top_ref;
        res = top->t;
        *top_ref = top->next;
        free(top);
        return res;
    }
}
```

```
    }
}

/* Helper function that allocates a new tNode with the
   given data and NULL left and right pointers. */
struct tNode* newtNode(int data)
{
    struct tNode* tNode = (struct tNode*)
                           malloc(sizeof(struct tNode));

    tNode->data = data;
    tNode->left = NULL;
    tNode->right = NULL;

    return(tNode);
}

/* Driver program to test above functions*/
int main()
{
    /* Constructed binary tree is
        1
       / \
      2   3
     / \
    4   5
    */
    struct tNode *root = newtNode(1);
    root->left = newtNode(2);
    root->right = newtNode(3);
    root->left->left = newtNode(4);
    root->left->right = newtNode(5);

    inOrder(root);

    getchar();
    return 0;
}
```

Java

```
// non-recursive java program for inorder traversal
import java.util.Stack;

/* Class containing left and right child of
current node and key value*/
class Node
{
```

```
int data;
Node left, right;

public Node(int item)
{
    data = item;
    left = right = null;
}
}

/* Class to print the inorder traversal */
class BinaryTree
{
    Node root;
    void inorder()
    {
        if (root == null)
            return;

        Stack<Node> s = new Stack<Node>();
        Node curr = root;

        // traverse the tree
        while (curr != null || s.size() > 0)
        {
            /* Reach the left most Node of the
            curr Node */
            while (curr != null)
            {
                /* place pointer to a tree node on
                the stack before traversing
                the node's left subtree */
                s.push(curr);
                curr = curr.left;
            }

            /* Current must be NULL at this point */
            curr = s.pop();

            System.out.print(curr.data + " ");

            /* we have visited the node and its
            left subtree. Now, it's right
            subtree's turn */
            curr = curr.right;
        }
    }
}
```



```
}

public static void main(String args[])
{

    /* creating a binary tree and entering
    the nodes */
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.inorder();
}
}
```

Python

```
# Python program to do inorder traversal without recursion

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Iterative function for inorder tree traversal
def inOrder(root):

    # Set current to root of binary tree
    current = root
    s = [] # initialize stack
    done = 0

    while(not done):

        # Reach the left most Node of the current Node
        if current is not None:

            # Place pointer to a tree node on the stack
            # before traversing the node's left subtree
            s.append(current)

            current = current.left
```

```
# BackTrack from the empty subtree and visit the Node
# at the top of the stack; however, if the stack is
# empty you are done
else:
    if(len(s) >0 ):
        current = s.pop()
        print current.data,

        # We have visited the node and its left
        # subtree. Now, it's right subtree's turn
        current = current.right

    else:
        done = 1

# Driver program to test above function

""" Constructed binary tree is
      1
     / \
    2   3
   / \
  4   5  """

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

inOrder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Time Complexity: $O(n)$

Output:

4 2 5 1 3

References:

<http://web.cs.wpi.edu/~cs2005/common/iterative.inorder>

<http://neural.cs.nthu.edu.tw/jang/courses/cs2351/slide/animation/Iterative%20Inorder%20Traversal.pps>

See [this post](#) for another approach of Inorder Tree Traversal without recursion and without stack!

Improved By : [Rishabh Jindal 2](#)

Source

<https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/>

Chapter 54

Interleave the first half of the queue with second half

Interleave the first half of the queue with second half - GeeksforGeeks

Given a queue of integers of even length, rearrange the elements by interleaving the first half of the queue with the second half of the queue.

Only a stack can be used as an auxiliary space.

Examples:

Input : 1 2 3 4

Output : 1 3 2 4

Input : 11 12 13 14 15 16 17 18 19 20

Output : 11 16 12 17 13 18 14 19 15 20

Following are the steps to solve the problem:

1. Push the first half elements of queue to stack.
2. Enqueue back the stack elements.
3. Dequeue the first half elements of the queue and enqueue them back.
4. Again push the first half elements into the stack.
5. Interleave the elements of queue and stack.

```
// C++ program to interleave the first half of the queue
// with the second half
#include <bits/stdc++.h>
using namespace std;

// Function to interleave the queue
void interLeaveQueue(queue<int>& q)
```

```
{
    // To check the even number of elements
    if (q.size() % 2 != 0)
        cout << "Input even number of integers." << endl;

    // Initialize an empty stack of int type
    stack<int> s;
    int halfSize = q.size() / 2;

    // Push first half elements into the stack
    // queue:16 17 18 19 20, stack: 15(T) 14 13 12 11
    for (int i = 0; i < halfSize; i++) {
        s.push(q.front());
        q.pop();
    }

    // enqueue back the stack elements
    // queue: 16 17 18 19 20 15 14 13 12 11
    while (!s.empty()) {
        q.push(s.top());
        s.pop();
    }

    // dequeue the first half elements of queue
    // and enqueue them back
    // queue: 15 14 13 12 11 16 17 18 19 20
    for (int i = 0; i < halfSize; i++) {
        q.push(q.front());
        q.pop();
    }

    // Again push the first half elements into the stack
    // queue: 16 17 18 19 20, stack: 11(T) 12 13 14 15
    for (int i = 0; i < halfSize; i++) {
        s.push(q.front());
        q.pop();
    }

    // interleave the elements of queue and stack
    // queue: 11 16 12 17 13 18 14 19 15 20
    while (!s.empty()) {
        q.push(s.top());
        s.pop();
        q.push(q.front());
        q.pop();
    }
}
```

```
// Driver program to test above function
int main()
{
    queue<int> q;
    q.push(11);
    q.push(12);
    q.push(13);
    q.push(14);
    q.push(15);
    q.push(16);
    q.push(17);
    q.push(18);
    q.push(19);
    q.push(20);
    interLeaveQueue(q);
    int length = q.size();
    for (int i = 0; i < length; i++) {
        cout << q.front() << " ";
        q.pop();
    }
    return 0;
}
```

Output:

11 16 12 17 13 18 14 19 15 20

Time complexity: $O(n)$.

Auxiliary Space : $O(n)$.

Source

<https://www.geeksforgeeks.org/interleave-first-half-queue-second-half/>

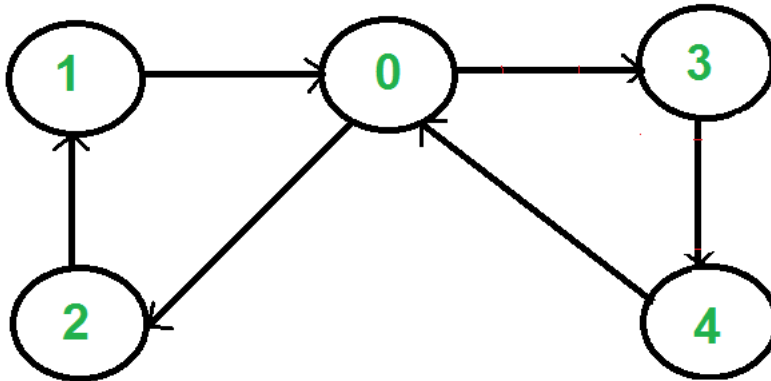
Chapter 55

Iterative Depth First Traversal of Graph

Iterative Depth First Traversal of Graph - GeeksforGeeks

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal (DFS) of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

For example, a DFS of below graph is “0 3 4 2 1”, other possible DFS is “0 2 1 3 4”.



We have discussed recursive implementation of DFS in previous in [previous post](#). In the post, iterative DFS is discussed. The recursive implementation uses function call stack. In iterative implementation, an explicit stack is used to hold visited vertices.

Below is implementation of Iterative DFS. *The implementation is similar to BFS, the only difference is queue is replaced by stack.*

C++

```
// An Iterative C++ program to do DFS traversal from
// a given source vertex. DFS(int s) traverses vertices
// reachable from s.
#include<bits/stdc++.h>
using namespace std;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // adjacency lists
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);    // to add an edge to graph
    void DFS(int s);    // prints all vertices in DFS manner
    // from a given source.
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);    // Add w to v's list.
}

// prints all not yet visited vertices reachable from s
void Graph::DFS(int s)
{
    // Initially mark all verices as not visited
    vector<bool> visited(V, false);

    // Create a stack for DFS
    stack<int> stack;

    // Push the current source node.
    stack.push(s);

    while (!stack.empty())
    {
        // Pop a vertex from stack and print it
        s = stack.top();
        stack.pop();
    }
}
```



```
// Stack may contain same vertex twice. So
// we need to print the popped item only
// if it is not visited.
if (!visited[s])
{
    cout << s << " ";
    visited[s] = true;
}

// Get all adjacent vertices of the popped vertex s
// If a adjacent has not been visited, then push it
// to the stack.
for (auto i = adj[s].begin(); i != adj[s].end(); ++i)
    if (!visited[*i])
        stack.push(*i);
}
}

// Driver program to test methods of graph class
int main()
{
    Graph g(5); // Total 5 vertices in graph
    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(1, 4);

    cout << "Following is Depth First Traversal\n";
    g.DFS(0);

    return 0;
}
```

Java

```
//An Iterative Java program to do DFS traversal from
//a given source vertex. DFS(int s) traverses vertices
//reachable from s.

import java.util.*;

public class GFG
{
    // This class represents a directed graph using adjacency
    // list representation
    static class Graph
    {
```

```
int V; //Number of Vertices

LinkedList<Integer>[] adj; // adjacency lists

//Constructor
Graph(int V)
{
    this.V = V;
    adj = new LinkedList[V];

    for (int i = 0; i < adj.length; i++)
        adj[i] = new LinkedList<Integer>();
}

//To add an edge to graph
void addEdge(int v, int w)
{
    adj[v].add(w); // Add w to v's list.
}

// prints all not yet visited vertices reachable from s
void DFS(int s)
{
    // Initially mark all vertices as not visited
    Vector<Boolean> visited = new Vector<Boolean>(V);
    for (int i = 0; i < V; i++)
        visited.add(false);

    // Create a stack for DFS
    Stack<Integer> stack = new Stack<>();

    // Push the current source node
    stack.push(s);

    while(stack.empty() == false)
    {
        // Pop a vertex from stack and print it
        s = stack.peek();
        stack.pop();

        // Stack may contain same vertex twice. So
        // we need to print the popped item only
        // if it is not visited.
        if(visited.get(s) == false)
        {
            System.out.print(s + " ");
            visited.set(s, true);
        }
    }
}
```

```
        }

        // Get all adjacent vertices of the popped vertex s
        // If a adjacent has not been visited, then push it
        // to the stack.
        Iterator<Integer> itr = adj[s].iterator();

        while (itr.hasNext())
        {
            int v = itr.next();
            if(!visited.get(v))
                stack.push(v);
        }
    }
}

// Driver program to test methods of graph class
public static void main(String[] args)
{
    // Total 5 vertices in graph
    Graph g = new Graph(5);

    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(1, 4);

    System.out.println("Following is the Depth First Traversal");
    g.DFS(0);
}
}
```

Output:

Following is Depth First Traversal
0 3 2 1 4

Note that the above implementation prints only vertices that are reachable from a given vertex. For example, if we remove edges 0-3 and 0-2, the above program would only print 0. To print all vertices of a graph, we need to call DFS for every vertex. Below is implementation for the same.

C++

```
// An Iterative C++ program to do DFS traversal from
// a given source vertex. DFS(int s) traverses vertices
// reachable from s.
#include<bits/stdc++.h>
using namespace std;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // adjacency lists
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);    // to add an edge to graph
    void DFS();    // prints all vertices in DFS manner

    // prints all not yet visited vertices reachable from s
    void DFSUtil(int s, vector<bool> &visited);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);    // Add w to v's list.
}

// prints all not yet visited vertices reachable from s
void Graph::DFSUtil(int s, vector<bool> &visited)
{
    // Create a stack for DFS
    stack<int> stack;

    // Push the current source node.
    stack.push(s);

    while (!stack.empty())
    {
        // Pop a vertex from stack and print it
        s = stack.top();
        stack.pop();

        // Stack may contain same vertex twice. So
```

```
// we need to print the popped item only
// if it is not visited.
if (!visited[s])
{
    cout << s << " ";
    visited[s] = true;
}

// Get all adjacent vertices of the popped vertex s
// If a adjacent has not been visited, then push it
// to the stack.
for (auto i = adj[s].begin(); i != adj[s].end(); ++i)
    if (!visited[*i])
        stack.push(*i);
}

// prints all vertices in DFS manner
void Graph::DFS()
{
    // Mark all the vertices as not visited
    vector<bool> visited(V, false);

    for (int i = 0; i < V; i++)
        if (!visited[i])
            DFSUtil(i, visited);
}

// Driver program to test methods of graph class
int main()
{
    Graph g(5); // Total 5 vertices in graph
    g.addEdge(1, 0);
    g.addEdge(2, 1);
    g.addEdge(3, 4);
    g.addEdge(4, 0);

    cout << "Following is Depth First Traversal\n";
    g.DFS();

    return 0;
}
```

Java

```
//An Iterative Java program to do DFS traversal from
//a given source vertex. DFS() traverses vertices
//reachable from s.
```

```
import java.util.*;

public class GFG
{
    // This class represents a directed graph using adjacency
    // list representation
    static class Graph
    {
        int V; //Number of Vertices

        LinkedList<Integer>[] adj; // adjacency lists

        //Constructor
        Graph(int V)
        {
            this.V = V;
            adj = new LinkedList[V];

            for (int i = 0; i < adj.length; i++)
                adj[i] = new LinkedList<Integer>();
        }

        //To add an edge to graph
        void addEdge(int v, int w)
        {
            adj[v].add(w); // Add w to v's list.
        }

        // prints all not yet visited vertices reachable from s
        void DFSUtil(int s, Vector<Boolean> visited)
        {
            // Create a stack for DFS
            Stack<Integer> stack = new Stack<>();

            // Push the current source node
            stack.push(s);

            while(stack.empty() == false)
            {
                // Pop a vertex from stack and print it
                s = stack.peek();
                stack.pop();

                // Stack may contain same vertex twice. So
                // we need to print the popped item only
                // if it is not visited.
            }
        }
    }
}
```

```
        if(visited.get(s) == false)
        {
            System.out.print(s + " ");
            visited.set(s, true);
        }

        // Get all adjacent vertices of the popped vertex s
        // If a adjacent has not been visited, then push it
        // to the stack.
        Iterator<Integer> itr = adj[s].iterator();

        while (itr.hasNext())
        {
            int v = itr.next();
            if(!visited.get(v))
                stack.push(v);
        }
    }
}

// prints all vertices in DFS manner
void DFS()
{
    Vector<Boolean> visited = new Vector<Boolean>(V);
    // Mark all the vertices as not visited
    for (int i = 0; i < V; i++)
        visited.add(false);

    for (int i = 0; i < V; i++)
        if (!visited.get(i))
            DFSUtil(i, visited);
}

// Driver program to test methods of graph class
public static void main(String[] args)
{
    Graph g = new Graph(5);
    g.addEdge(1, 0);
    g.addEdge(2, 1);
    g.addEdge(3, 4);
    g.addEdge(4, 0);

    System.out.println("Following is Depth First Traversal");
    g.DFS();
}
}
```

Output:

Following is Depth First Traversal
0 1 2 3 4

Like recursive traversal, time complexity of iterative implementation is $O(V + E)$.

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/iterative-depth-first-traversal/>

Chapter 56

Iterative Postorder Traversal | Set 1 (Using Two Stacks)

Iterative Postorder Traversal | Set 1 (Using Two Stacks) - GeeksforGeeks

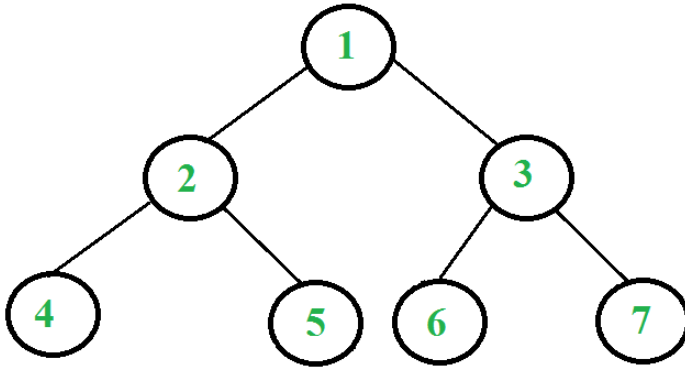
We have discussed [iterative inorder](#) and [iterative preorder](#) traversals. In this post, iterative postorder traversal is discussed, which is more complex than the other two traversals (due to its nature of non-[tail recursion](#), there is an extra statement after the final recursive call to itself). Postorder traversal can easily be done using two stacks, though. The idea is to push reverse postorder traversal to a stack. Once we have the reversed postorder traversal in a stack, we can just pop all items one by one from the stack and print them; this order of printing will be in postorder because of the LIFO property of stacks. Now the question is, how to get reversed postorder elements in a stack – the second stack is used for this purpose. For example, in the following tree, we need to get 1, 3, 7, 6, 2, 5, 4 in a stack. If take a closer look at this sequence, we can observe that this sequence is very similar to the preorder traversal. The only difference is that the right child is visited before left child, and therefore the sequence is “root right left” instead of “root left right”. So, we can do something like [iterative preorder traversal](#) with the following differences:

- a) Instead of printing an item, we push it to a stack.
- b) We push the left subtree before the right subtree.

Following is the complete algorithm. After step 2, we get the reverse of a postorder traversal in the second stack. We use the first stack to get the correct order.

1. Push root to first stack.
2. Loop while first stack is not empty
 - 2.1 Pop a node from first stack and push it to second stack
 - 2.2 Push left and right children of the popped node to first stack
3. Print contents of second stack

Let us consider the following tree



Following are the steps to print postorder traversal of the above tree using two stacks.

1. Push 1 to first stack.
First stack: 1
Second stack: Empty
2. Pop 1 from first stack and push it to second stack.
Push left and right children of 1 to first stack
First stack: 2, 3
Second stack: 1
3. Pop 3 from first stack and push it to second stack.
Push left and right children of 3 to first stack
First stack: 2, 6, 7
Second stack: 1, 3
4. Pop 7 from first stack and push it to second stack.
First stack: 2, 6
Second stack: 1, 3, 7
5. Pop 6 from first stack and push it to second stack.
First stack: 2
Second stack: 1, 3, 7, 6
6. Pop 2 from first stack and push it to second stack.
Push left and right children of 2 to first stack
First stack: 4, 5
Second stack: 1, 3, 7, 6, 2
7. Pop 5 from first stack and push it to second stack.
First stack: 4
Second stack: 1, 3, 7, 6, 2, 5
8. Pop 4 from first stack and push it to second stack.

First stack: Empty
Second stack: 1, 3, 7, 6, 2, 5, 4

The algorithm stops here since there are no more items in the first stack. Observe that the contents of second stack are in postorder fashion. Print them.

Following is C implementation of iterative postorder traversal using two stacks.

C

```
#include <stdio.h>
#include <stdlib.h>

// Maximum stack size
#define MAX_SIZE 100

// A tree node
struct Node {
    int data;
    struct Node *left, *right;
};

// Stack type
struct Stack {
    int size;
    int top;
    struct Node** array;
};

// A utility function to create a new tree node
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// A utility function to create a stack of given size
struct Stack* createStack(int size)
{
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->size = size;
    stack->top = -1;
    stack->array = (struct Node**)malloc(stack->size * sizeof(struct Node*));
    return stack;
}
```

```
// BASIC OPERATIONS OF STACK
int isFull(struct Stack* stack)
{
    return stack->top - 1 == stack->size;
}

int isEmpty(struct Stack* stack)
{
    return stack->top == -1;
}

void push(struct Stack* stack, struct Node* node)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = node;
}

struct Node* pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}

// An iterative function to do post order traversal of a given binary tree
void postOrderIterative(struct Node* root)
{
    if (root == NULL)
        return;

    // Create two stacks
    struct Stack* s1 = createStack(MAX_SIZE);
    struct Stack* s2 = createStack(MAX_SIZE);

    // push root to first stack
    push(s1, root);
    struct Node* node;

    // Run while first stack is not empty
    while (!isEmpty(s1)) {
        // Pop an item from s1 and push it to s2
        node = pop(s1);
        push(s2, node);

        // Push left and right children of removed item to s1
        if (node->left)
```

```
        push(s1, node->left);
    if (node->right)
        push(s1, node->right);
}

// Print all elements of second stack
while (!isEmpty(s2)) {
    node = pop(s2);
    printf("%d ", node->data);
}

}

// Driver program to test above functions
int main()
{
    // Let us construct the tree shown in above figure
    struct Node* root = NULL;
    root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    postOrderIterative(root);

    return 0;
}
```

Java

```
// Java program for iterative post
// order using two stacks

import java.util.*;
public class IterativePostorder {

    static class node {
        int data;
        node left, right;

        public node(int data)
        {
            this.data = data;
        }
    }
}
```

```
// Two stacks as used in explanation
static Stack<node> s1, s2;

static void postOrderIterative(node root)
{
    // Create two stacks
    s1 = new Stack<>();
    s2 = new Stack<>();

    if (root == null)
        return;

    // push root to first stack
    s1.push(root);

    // Run while first stack is not empty
    while (!s1.isEmpty()) {
        // Pop an item from s1 and push it to s2
        node temp = s1.pop();
        s2.push(temp);

        // Push left and right children of
        // removed item to s1
        if (temp.left != null)
            s1.push(temp.left);
        if (temp.right != null)
            s1.push(temp.right);
    }

    // Print all elements of second stack
    while (!s2.isEmpty()) {
        node temp = s2.pop();
        System.out.print(temp.data + " ");
    }
}

public static void main(String[] args)
{
    // Let us construct the tree
    // shown in above figure

    node root = null;
    root = new node(1);
    root.left = new node(2);
    root.right = new node(3);
    root.left.left = new node(4);
    root.left.right = new node(5);
    root.right.left = new node(6);
}
```

```
        root.right.right = new node(7);

        postOrderIterative(root);
    }
}

// This code is contributed by Rishabh Mahrsee
```

Python

```
# Python program for iterative postorder traversal using
# two stacks

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# An iterative function to do postorder traversal of a
# given binary tree
def postOrderIterative(root):

    if root is None:
        return

    # Create two stacks
    s1 = []
    s2 = []

    # Push root to first stack
    s1.append(root)

    # Run while first stack is not empty
    while s1:

        # Pop an item from s1 and append it to s2
        node = s1.pop()
        s2.append(node)

        # Push left and right children of removed item to s1
        if node.left:
            s1.append(node.left)
        if node.right:
            s1.append(node.right)
```

```
        # Print all elements of second stack
    while s2:
        node = s2.pop()
        print node.data,

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)
postOrderIterative(root)
```

Output:

4 5 2 6 7 3 1

Following is an overview of the above post.

Iterative preorder traversal can be easily implemented using two stacks. The first stack is used to get the reverse postorder traversal. The steps to get a reverse postorder are similar to [iterative preorder](#).

You may also like to see [a method which uses only one stack](#).

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : [IshitaTripathi](#)

Source

<https://www.geeksforgeeks.org/iterative-postorder-traversal/>

Chapter 57

Iterative Postorder Traversal | Set 2 (Using One Stack)

Iterative Postorder Traversal | Set 2 (Using One Stack) - GeeksforGeeks

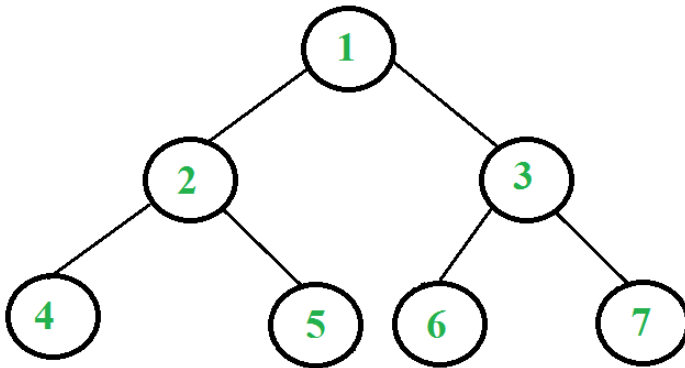
We have discussed a simple [iterative postorder traversal using two stacks](#) in the previous post. In this post, an approach with only one stack is discussed.

The idea is to move down to leftmost node using left pointer. While moving down, push root and root's right child to stack. Once we reach leftmost node, print it if it doesn't have a right child. If it has a right child, then change root so that the right child is processed before.

Following is detailed algorithm.

- 1.1 Create an empty stack
- 2.1 Do following while root is not NULL
 - a) Push root's right child and then root to stack.
 - b) Set root as root's left child.
- 2.2 Pop an item from stack and set it as root.
 - a) If the popped item has a right child and the right child is at top of stack, then remove the right child from stack, push the root back and set root as root's right child.
 - b) Else print root's data and set root as NULL.
- 2.3 Repeat steps 2.1 and 2.2 while stack is not empty.

Let us consider the following tree



Following are the steps to print postorder traversal of the above tree using one stack.

1. Right child of 1 exists.
Push 3 to stack. Push 1 to stack. Move to left child.
Stack: 3, 1
2. Right child of 2 exists.
Push 5 to stack. Push 2 to stack. Move to left child.
Stack: 3, 1, 5, 2
3. Right child of 4 doesn't exist. '
Push 4 to stack. Move to left child.
Stack: 3, 1, 5, 2, 4
4. Current node is NULL.
Pop 4 from stack. Right child of 4 doesn't exist.
Print 4. Set current node to NULL.
Stack: 3, 1, 5, 2
5. Current node is NULL.
Pop 2 from stack. Since right child of 2 equals stack top element,
pop 5 from stack. Now push 2 to stack.
Move current node to right child of 2 i.e. 5
Stack: 3, 1, 2
6. Right child of 5 doesn't exist. Push 5 to stack. Move to left child.
Stack: 3, 1, 2, 5
7. Current node is NULL. Pop 5 from stack. Right child of 5 doesn't exist.
Print 5. Set current node to NULL.
Stack: 3, 1, 2
8. Current node is NULL. Pop 2 from stack.
Right child of 2 is not equal to stack top element.

Print 2. Set current node to NULL.

Stack: 3, 1

9. Current node is NULL. Pop 1 from stack.

Since right child of 1 equals stack top element, pop 3 from stack.

Now push 1 to stack. Move current node to right child of 1 i.e. 3

Stack: 1

10. Repeat the same as above steps and Print 6, 7 and 3.

Pop 1 and Print 1.

C

```
// C program for iterative postorder traversal using one stack
#include <stdio.h>
#include <stdlib.h>

// Maximum stack size
#define MAX_SIZE 100

// A tree node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Stack type
struct Stack
{
    int size;
    int top;
    struct Node* *array;
};

// A utility function to create a new tree node
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// A utility function to create a stack of given size
struct Stack* createStack(int size)
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
```

```
    stack->size = size;
    stack->top = -1;
    stack->array = (struct Node**) malloc(stack->size * sizeof(struct Node*));
    return stack;
}

// BASIC OPERATIONS OF STACK
int isFull(struct Stack* stack)
{ return stack->top - 1 == stack->size; }

int isEmpty(struct Stack* stack)
{ return stack->top == -1; }

void push(struct Stack* stack, struct Node* node)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = node;
}

struct Node* pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}

struct Node* peek(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top];
}

// An iterative function to do postorder traversal of a given binary tree
void postOrderIterative(struct Node* root)
{
    // Check for empty tree
    if (root == NULL)
        return;

    struct Stack* stack = createStack(MAX_SIZE);
    do
    {
        // Move to leftmost node
        while (root)
        {
            // Push root's right child and then root to stack.

```

```
        if (root->right)
            push(stack, root->right);
        push(stack, root);

        // Set root as root's left child
        root = root->left;
    }

    // Pop an item from stack and set it as root
    root = pop(stack);

    // If the popped item has a right child and the right child is not
    // processed yet, then make sure right child is processed before root
    if (root->right && peek(stack) == root->right)
    {
        pop(stack); // remove right child from stack
        push(stack, root); // push root back to stack
        root = root->right; // change root so that the right
                           // child is processed next
    }
    else // Else print root's data and set root as NULL
    {
        printf("%d ", root->data);
        root = NULL;
    }
} while (!isEmpty(stack));
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree shown in above figure
    struct Node* root = NULL;
    root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    printf("Post order traversal of binary tree is :\n");
    printf("[");
    postOrderIterative(root);
    printf("]");

    return 0;
}
```

Java

```
// A java program for iterative postorder traversal using stack

import java.util.ArrayList;
import java.util.Stack;

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right;
    }
}

class BinaryTree
{
    Node root;
    ArrayList<Integer> list = new ArrayList<Integer>();

    // An iterative function to do postorder traversal
    // of a given binary tree
    ArrayList<Integer> postOrderIterative(Node node)
    {
        Stack<Node> S = new Stack<Node>();

        // Check for empty tree
        if (node == null)
            return list;
        S.push(node);
        Node prev = null;
        while (!S.isEmpty())
        {
            Node current = S.peek();

            /* go down the tree in search of a leaf and if so process it
            and pop stack otherwise move down */
            if (prev == null || prev.left == current ||
                prev.right == current)
            {
                if (current.left != null)
                    S.push(current.left);
                else if (current.right != null)
```

```
        S.push(current.right);
    else
    {
        S.pop();
        list.add(current.data);
    }

    /* go up the tree from left node, if the child is right
       push it onto stack otherwise process parent and pop
       stack */
}
else if (current.left == prev)
{
    if (current.right != null)
        S.push(current.right);
    else
    {
        S.pop();
        list.add(current.data);
    }

    /* go up the tree from right node and after coming back
       from right node process parent and pop stack */
}
else if (current.right == prev)
{
    S.pop();
    list.add(current.data);
}

prev = current;
}

return list;
}

// Driver program to test above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    // Let us create trees shown in above diagram
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.left = new Node(6);
```

```
        tree.root.right.right = new Node(7);

        ArrayList<Integer> mylist = tree.postOrderIterative(tree.root);

        System.out.println("Post order traversal of binary tree is :");
        System.out.println(mylist);
    }
}

// This code has been contributed by Mayank Jaiswal
```

Python

```
# Python program for iterative postorder traversal
# using one stack

# Stores the answer
ans = []

# A Binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

def peek(stack):
    if len(stack) > 0:
        return stack[-1]
    return None

# A iterative function to do postorder traversal of
# a given binary tree
def postOrderIterative(root):

    # Check for empty tree
    if root is None:
        return

    stack = []

    while(True):

        while (root):
            # Push root's right child and then root to stack
            if root.right is not None:
                stack.append(root.right)
```



```
stack.append(root)

# Set root as root's left child
root = root.left

# Pop an item from stack and set it as root
root = stack.pop()

# If the popped item has a right child and the
# right child is not processed yet, then make sure
# right child is processed before root
if (root.right is not None and
    peek(stack) == root.right):
    stack.pop() # Remove right child from stack
    stack.append(root) # Push root back to stack
    root = root.right # change root so that the
                    # right child is processed next

# Else print root's data and set root as None
else:
    ans.append(root.data)
    root = None

if (len(stack) <= 0):
    break

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)

print "Post Order traversal of binary tree is"
postOrderIterative(root)
print ans

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Post Order traversal of binary tree is
[4, 5, 2, 6, 7, 3, 1]
```

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/iterative-postorder-traversal-using-stack/>

Chapter 58

Iterative Tower of Hanoi

Iterative Tower of Hanoi - GeeksforGeeks

Tower of Hanoi is a mathematical puzzle. It consists of three poles and a number of disks of different sizes which can slide onto any poles. The puzzle starts with the disk in a neat stack in ascending order of size in one pole, the smallest at the top thus making a conical shape. The objective of the puzzle is to move all the disks from one pole (say 'source pole') to another pole (say 'destination pole') with the help of third pole (say auxiliary pole).

The puzzle has the following two rules:

1. You can't place a larger disk onto smaller disk
2. Only one disk can be moved at a time

We've already discussed [recursive solution for Tower of Hanoi](#). We have also seen that, for n disks, total $2^n - 1$ moves are required.

Iterative Algorithm:

1. Calculate the total number of moves required i.e. " $\text{pow}(2, n) - 1$ " here n is number of disks.
2. If number of disks (i.e. n) is even then interchange destination pole and auxiliary pole.
3. for $i = 1$ to total number of moves:
 - if $i \% 3 == 1$:
legal movement of top disk between source pole and destination pole
 - if $i \% 3 == 2$:
legal movement top disk between source pole and auxiliary pole
 - if $i \% 3 == 0$:
legal movement top disk between auxiliary pole and destination pole

Example:

Let us understand with a simple example with 3 disks:

So, total number of moves required = 7

S A D

When i = 1, (i % 3 == 1) legal movement between 'S' and 'D'

When i = 2, (i % 3 == 2) legal movement between 'S' and 'A'

When i = 3, (i % 3 == 0) legal movement between 'A' and 'D' ,

When i = 4, (i % 3 == 1) legal movement between 'S' and 'D'

When i = 5, (i % 3 == 2) legal movement between 'S' and 'A'

When i = 6, (i % 3 == 0) legal movement between 'A' and 'D'

When i = 7, (i % 3 == 1) legal movement between 'S' and 'D'

So, after all these destination pole contains all the in order of size.

After observing above iterations, we can think that after a disk other than the smallest disk is moved, the next disk to be moved must be the smallest disk because it is the top disk resting on the spare pole and there are no other choices to move a disk.

C

```
// C Program for Iterative Tower of Hanoi
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a stack
struct Stack
{
    unsigned capacity;
    int top;
```

```
    int *array;
};

// function to create a stack of given capacity.
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack =
        (struct Stack*) malloc(sizeof(struct Stack));
    stack -> capacity = capacity;
    stack -> top = -1;
    stack -> array =
        (int*) malloc(stack -> capacity * sizeof(int));
    return stack;
}

// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{
    return (stack->top == stack->capacity - 1);
}

// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{
    return (stack->top == -1);
}

// Function to add an item to stack. It increases
// top by 1
void push(struct Stack *stack, int item)
{
    if (isFull(stack))
        return;
    stack -> array[++stack -> top] = item;
}

// Function to remove an item from stack. It
// decreases top by 1
int pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack -> array[stack -> top--];
}

// Function to implement legal movement between
// two poles
void moveDisksBetweenTwoPoles(struct Stack *src,
```

```

        struct Stack *dest, char s, char d)
{
    int pole1TopDisk = pop(src);
    int pole2TopDisk = pop(dest);

    // When pole 1 is empty
    if (pole1TopDisk == INT_MIN)
    {
        push(src, pole2TopDisk);
        moveDisk(d, s, pole2TopDisk);
    }

    // When pole2 pole is empty
    else if (pole2TopDisk == INT_MIN)
    {
        push(dest, pole1TopDisk);
        moveDisk(s, d, pole1TopDisk);
    }

    // When top disk of pole1 > top disk of pole2
    else if (pole1TopDisk > pole2TopDisk)
    {
        push(src, pole1TopDisk);
        push(src, pole2TopDisk);
        moveDisk(d, s, pole2TopDisk);
    }

    // When top disk of pole1 < top disk of pole2
    else
    {
        push(dest, pole2TopDisk);
        push(dest, pole1TopDisk);
        moveDisk(s, d, pole1TopDisk);
    }
}

//Function to show the movement of disks
void moveDisk(char fromPeg, char toPeg, int disk)
{
    printf("Move the disk %d from \'%c\' to \'%c\'\n",
        disk, fromPeg, toPeg);
}

//Function to implement TOH puzzle
void tohIterative(int num_of_disks, struct Stack
    *src, struct Stack *aux,
    struct Stack *dest)
{

```

```
int i, total_num_of_moves;
char s = 'S', d = 'D', a = 'A';

//If number of disks is even, then interchange
//destination pole and auxiliary pole
if (num_of_disks % 2 == 0)
{
    char temp = d;
    d = a;
    a = temp;
}
total_num_of_moves = pow(2, num_of_disks) - 1;

//Larger disks will be pushed first
for (i = num_of_disks; i >= 1; i--)
    push(src, i);

for (i = 1; i <= total_num_of_moves; i++)
{
    if (i % 3 == 1)
        moveDisksBetweenTwoPoles(src, dest, s, d);

    else if (i % 3 == 2)
        moveDisksBetweenTwoPoles(src, aux, s, a);

    else if (i % 3 == 0)
        moveDisksBetweenTwoPoles(aux, dest, a, d);
}
}

// Driver Program
int main()
{
    // Input: number of disks
    unsigned num_of_disks = 3;

    struct Stack *src, *dest, *aux;

    // Create three stacks of size 'num_of_disks'
    // to hold the disks
    src = createStack(num_of_disks);
    aux = createStack(num_of_disks);
    dest = createStack(num_of_disks);

    tohIterative(num_of_disks, src, aux, dest);
    return 0;
}
```

Java

```
// Java program for iterative
// Tower of Hanoi

public class TOH
{
    // A structure to represent a stack
    class Stack
    {
        int capacity;
        int top;
        int array[];
    }

    // function to create a stack of given capacity.
    Stack createStack(int capacity)
    {
        Stack stack=new Stack();
        stack.capacity = capacity;
        stack.top = -1;
        stack.array = new int[capacity];
        return stack;
    }

    // Stack is full when top is equal to the last index
    boolean isFull(Stack stack)
    {
        return (stack.top == stack.capacity - 1);
    }

    // Stack is empty when top is equal to -1
    boolean isEmpty(Stack stack)
    {
        return (stack.top == -1);
    }

    // Function to add an item to stack. It increases
    // top by 1
    void push(Stack stack,int item)
    {
        if(isFull(stack))
            return;
        stack.array[++stack.top] = item;
    }

    // Function to remove an item from stack. It
    // decreases top by 1
}
```



```
int pop(Stack stack)
{
    if(isEmpty(stack))
        return Integer.MIN_VALUE;
    return stack.array[stack.top--];
}

// Function to implement legal movement between
// two poles
void moveDisksBetweenTwoPoles(Stack src, Stack dest,
                               char s, char d)
{
    int pole1TopDisk = pop(src);
    int pole2TopDisk = pop(dest);

    // When pole 1 is empty
    if (pole1TopDisk == Integer.MIN_VALUE)
    {
        push(src, pole2TopDisk);
        moveDisk(d, s, pole2TopDisk);
    }
    // When pole2 pole is empty
    else if (pole2TopDisk == Integer.MIN_VALUE)
    {
        push(dest, pole1TopDisk);
        moveDisk(s, d, pole1TopDisk);
    }
    // When top disk of pole1 > top disk of pole2
    else if (pole1TopDisk > pole2TopDisk)
    {
        push(src, pole1TopDisk);
        push(src, pole2TopDisk);
        moveDisk(d, s, pole2TopDisk);
    }
    // When top disk of pole1 < top disk of pole2
    else
    {
        push(dest, pole2TopDisk);
        push(dest, pole1TopDisk);
        moveDisk(s, d, pole1TopDisk);
    }
}

// Function to show the movement of disks
void moveDisk(char fromPeg, char toPeg, int disk)
{
    System.out.println("Move the disk "+disk +
                       " from "+fromPeg+" to "+toPeg);
}
```

```
}

// Function to implement TOH puzzle
void tohIterative(int num_of_disks, Stack
                  src, Stack aux, Stack dest)
{
    int i, total_num_of_moves;
    char s = 'S', d = 'D', a = 'A';

    // If number of disks is even, then interchange
    // destination pole and auxiliary pole
    if (num_of_disks % 2 == 0)
    {
        char temp = d;
        d = a;
        a = temp;
    }
    total_num_of_moves = (int) (Math.pow(2, num_of_disks) - 1);

    // Larger disks will be pushed first
    for (i = num_of_disks; i >= 1; i--)
        push(src, i);

    for (i = 1; i <= total_num_of_moves; i++)
    {
        if (i % 3 == 1)
            moveDisksBetweenTwoPoles(src, dest, s, d);

        else if (i % 3 == 2)
            moveDisksBetweenTwoPoles(src, aux, s, a);

        else if (i % 3 == 0)
            moveDisksBetweenTwoPoles(aux, dest, a, d);
    }
}

// Driver Program to test above functions
public static void main(String[] args)
{
    // Input: number of disks
    int num_of_disks = 3;

    TOH ob = new TOH();
    Stack src, dest, aux;

    // Create three stacks of size 'num_of_disks'
    // to hold the disks
```

```
        src = ob.createStack(num_of_disks);
        dest = ob.createStack(num_of_disks);
        aux = ob.createStack(num_of_disks);

        ob.tohIterative(num_of_disks, src, aux, dest);
    }
}

// This code is Contributed by Sumit Ghosh
```

Output:

```
Move the disk 1 from 'S' to 'D'
Move the disk 2 from 'S' to 'A'
Move the disk 1 from 'D' to 'A'
Move the disk 3 from 'S' to 'D'
Move the disk 1 from 'A' to 'S'
Move the disk 2 from 'A' to 'D'
Move the disk 1 from 'S' to 'D'
```

Related Articles

- [Recursive Functions](#)
- [Tail recursion](#)
- [Quiz on Recursion](#)

References:

http://en.wikipedia.org/wiki/Tower_of_Hanoi#Iterative_solution

This article is contributed by **Anand Barnwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : [VIKASGUPTA1127](#)

Source

<https://www.geeksforgeeks.org/iterative-tower-of-hanoi/>

Chapter 59

Iterative method to find ancestors of a given binary tree

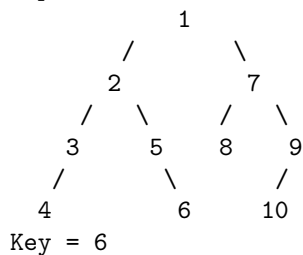
Iterative method to find ancestors of a given binary tree - GeeksforGeeks

Given a binary tree, print all the ancestors of a particular key existing in the tree without using recursion.

Here we will be discussing the c++ implementation for the above problem.

Examples:

Input :



Output : 5 2 1

Ancestors of 6 are 5, 2 and 1.

The idea is to use [iterative postorder traversal](#) of given binary tree.

```
// C++ program to print all ancestors of a given key
#include <bits/stdc++.h>
using namespace std;

// Structure for a tree node
```

```
struct Node {
    int data;
    struct Node* left, *right;
};

// A utility function to create a new tree node
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Iterative Function to print all ancestors of a
// given key
void printAncestors(struct Node* root, int key)
{
    if (root == NULL)
        return;

    // Create a stack to hold ancestors
    stack<struct Node*> st;

    // Traverse the complete tree in postorder way till
    // we find the key
    while (1) {

        // Traverse the left side. While traversing, push
        // the nodes into the stack so that their right
        // subtrees can be traversed later
        while (root && root->data != key) {
            st.push(root); // push current node
            root = root->left; // move to next node
        }

        // If the node whose ancestors are to be printed
        // is found, then break the while loop.
        if (root && root->data == key)
            break;

        // Check if right sub-tree exists for the node at top
        // If not then pop that node because we don't need
        // this node any more.
        if (st.top()->right == NULL) {
            root = st.top();
            st.pop();
        }
    }
}
```

```
        // If the popped node is right child of top,
        // then remove the top as well. Left child of
        // the top must have processed before.
        while (!st.empty() && st.top()->right == root) {
            root = st.top();
            st.pop();
        }
    }

    // if stack is not empty then simply set the root
    // as right child of top and start traversing right
    // sub-tree.
    root = st.empty() ? NULL : st.top()->right;
}

// If stack is not empty, print contents of stack
// Here assumption is that the key is there in tree
while (!st.empty()) {
    cout << st.top()->data << " ";
    st.pop();
}

// Driver program to test above functions
int main()
{
    // Let us construct a binary tree
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(7);
    root->left->left = newNode(3);
    root->left->right = newNode(5);
    root->right->left = newNode(8);
    root->right->right = newNode(9);
    root->left->left->left = newNode(4);
    root->left->right->right = newNode(6);
    root->right->right->left = newNode(10);

    int key = 6;
    printAncestors(root, key);

    return 0;
}
```

Output:

5 2 1

Source

<https://www.geeksforgeeks.org/iterative-method-to-find-ancestors-of-a-given-binary-tree/>

Chapter 60

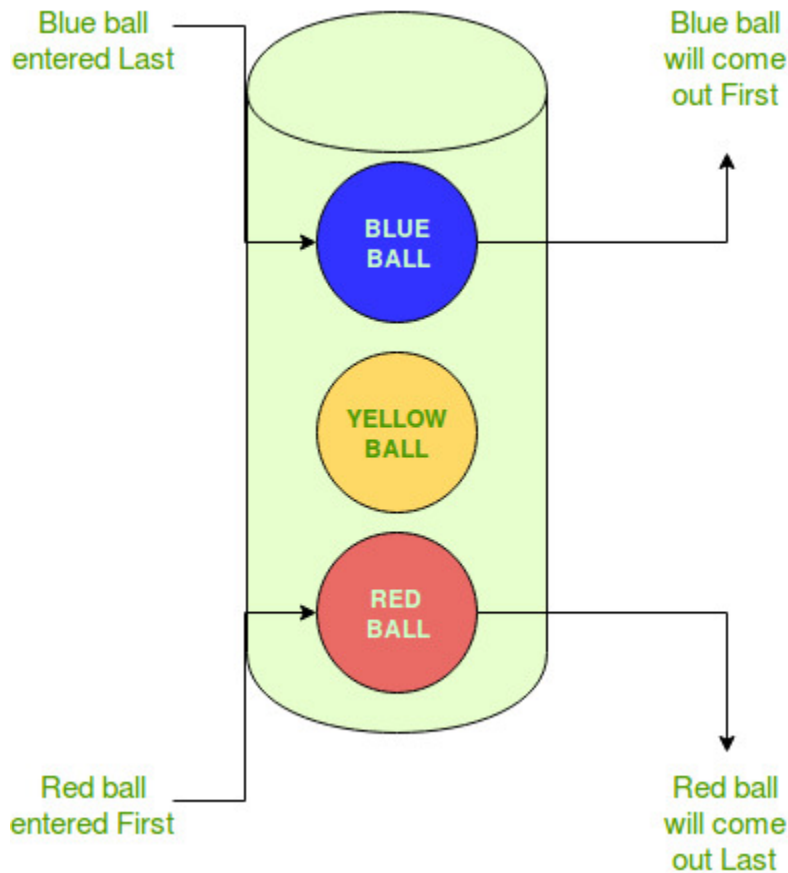
LIFO (Last-In-First-Out) approach in Programming

LIFO (Last-In-First-Out) approach in Programming - GeeksforGeeks

Prerequisites – [FIFO \(First-In-First-Out\) approach in Programming](#), [FIFO vs LIFO approach in Programming](#)

LIFO is an abbreviation for **last in, first out**. It is a method for handling data structures where the **first element** is processed last and the **last element** is processed first.

Real life example:



In this example, following things are to be considered:

- There is a bucket which holds balls.
- Different types of balls are entered in the bucket.
- The ball to enter the bucket last, will be taken out first.
- The ball entering the bucket next to last will be taken out after the ball above it (the newer one).
- In this way, the ball entering the bucket first will leave the bucket last.
- Therefore, the Last ball (Blue) to enter the bucket gets removed first and the First ball (Red) to enter the bucket gets removed last.

This is known as Last-In-First-Out approach or LIFO.

Where is LIFO used:

1. **Data Structures –**
Certain data structures like Stacks and other variants of Stacks uses LIFO approach for processing data.
2. **Extracting latest information –**
Sometimes computers use LIFO when data is extracted from an array or data buffer.

When it is required to get the most recent information entered, the LIFO approach is used.

Program Examples for LIFO –

Using Stack data structure:

```
// Java program to demonstrate
// working of LIFO
// using Stack in Java

import java.io.*;
import java.util.*;

class GFG {
    // Pushing element on the top of the stack
    static void stack_push(Stack<Integer> stack)
    {
        for (int i = 0; i < 5; i++) {
            stack.push(i);
        }
    }

    // Popping element from the top of the stack
    static void stack_pop(Stack<Integer> stack)
    {
        System.out.println("Pop :");

        for (int i = 0; i < 5; i++) {
            Integer y = (Integer)stack.pop();
            System.out.println(y);
        }
    }

    // Displaying element on the top of the stack
    static void stack_peek(Stack<Integer> stack)
    {
        Integer element = (Integer)stack.peek();
        System.out.println("Element on stack top : " + element);
    }

    // Searching element in the stack
    static void stack_search(Stack<Integer> stack, int element)
    {
        Integer pos = (Integer)stack.search(element);

        if (pos == -1)
            System.out.println("Element not found");
        else
```

```
        System.out.println("Element is found at position " + pos);
    }

    public static void main(String[] args)
    {
        Stack<Integer> stack = new Stack<Integer>();

        stack_push(stack);
        stack_pop(stack);
        stack_push(stack);
        stack_peek(stack);
        stack_search(stack, 2);
        stack_search(stack, 6);
    }
}
```

Output:

```
Pop:
4
3
2
1
0
Element on stack top : 4
Element is found at position 3
Element not found
```

Source

<https://www.geeksforgeeks.org/lifo-last-in-first-out-approach-in-programming/>

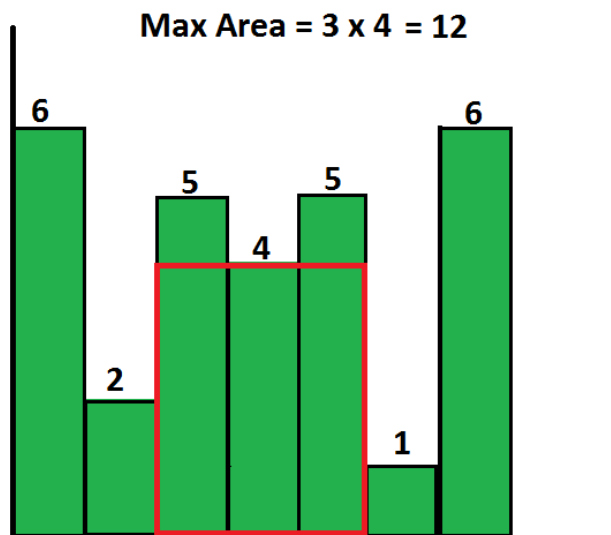
Chapter 61

Largest Rectangular Area in a Histogram | Set 1

Largest Rectangular Area in a Histogram | Set 1 - GeeksforGeeks

Find the largest rectangular area possible in a given histogram where the largest rectangle can be made of a number of contiguous bars. For simplicity, assume that all bars have same width and the width is 1 unit.

For example, consider the following histogram with 7 bars of heights {6, 2, 5, 4, 5, 2, 6}. The largest possible rectangle possible is 12 (see the below figure, the max area rectangle is highlighted in red)



A **simple solution** is to one by one consider all bars as starting points and calculate area of

all rectangles starting with every bar. Finally return maximum of all possible areas. Time complexity of this solution would be $O(n^2)$.

We can use **Divide and Conquer** to solve this in $O(n \log n)$ time. The idea is to find the minimum value in the given array. Once we have index of the minimum value, the max area is maximum of following three values.

- a) Maximum area in left side of minimum value (Not including the min value)
- b) Maximum area in right side of minimum value (Not including the min value)
- c) Number of bars multiplied by minimum value.

The areas in left and right of minimum value bar can be calculated recursively. If we use linear search to find the minimum value, then the worst case time complexity of this algorithm becomes $O(n^2)$. In worst case, we always have $(n-1)$ elements in one side and 0 elements in other side and if the finding minimum takes $O(n)$ time, we get the recurrence similar to worst case of Quick Sort.

How to find the minimum efficiently? **Range Minimum Query using Segment Tree** can be used for this. We build segment tree of the given histogram heights. Once the segment tree is built, all **range minimum queries take $O(\log n)$ time**. So over all complexity of the algorithm becomes.

Overall Time = Time to build Segment Tree + Time to recursively find maximum area

Time to build segment tree is $O(n)$. Let the time to recursively find max area be $T(n)$. It can be written as following.

$$T(n) = O(\log n) + T(n-1)$$

The solution of above recurrence is $O(n \log n)$. So overall time is $O(n) + O(n \log n)$ which is $O(n \log n)$.

Following is C++ implementation of the above algorithm.

```
// A Divide and Conquer Program to find maximum rectangular area in a histogram
#include <math.h>
#include <limits.h>
#include <iostream>
using namespace std;

// A utility function to find minimum of three integers
int max(int x, int y, int z)
{ return max(max(x, y), z); }

// A utility function to get minimum of two numbers in hist[]
int minVal(int *hist, int i, int j)
{
    if (i == -1) return j;
    if (j == -1) return i;
    return (hist[i] < hist[j])? i : j;
}

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e)
{ return s + (e - s)/2; }
```

```
/* A recursive function to get the index of minimum value in a given range of
indexes. The following are parameters for this function.

hist  --> Input array for which segment tree is built
st    --> Pointer to segment tree
index --> Index of current node in the segment tree. Initially 0 is
          passed as root is always at index 0
ss & se --> Starting and ending indexes of the segment represented by
            current node, i.e., st[index]
qs & qe --> Starting and ending indexes of query range */
int RMQUtil(int *hist, int *st, int ss, int se, int qs, int qe, int index)
{
    // If segment of this node is a part of given range, then return the
    // min of the segment
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return -1;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return minVal(hist, RMQUtil(hist, st, ss, mid, qs, qe, 2*index+1),
                  RMQUtil(hist, st, mid+1, se, qs, qe, 2*index+2));
}

// Return index of minimum element in range from index qs (query start) to
// qe (query end). It mainly uses RMQUtil()
int RMQ(int *hist, int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        cout << "Invalid Input";
        return -1;
    }

    return RMQUtil(hist, st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for hist[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int hist[], int ss, int se, int *st, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
```

```
    if (ss == se)
        return (st[si] = ss);

    // If there are more than one elements, then recur for left and
    // right subtrees and store the minimum of two values in this node
    int mid = getMid(ss, se);
    st[si] = minVal(hist, constructSTUtil(hist, ss, mid, st, si*2+1),
                   constructSTUtil(hist, mid+1, se, st, si*2+2));
    return st[si];
}

/* Function to construct segment tree from given array. This function
   allocates memory for segment tree and calls constructSTUtil() to
   fill the allocated memory */
int *constructST(int hist[], int n)
{
    // Allocate memory for segment tree
    int x = (int)(ceil(log2(n))); //Height of segment tree
    int max_size = 2*(int)pow(2, x) - 1; //Maximum size of segment tree
    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(hist, 0, n-1, st, 0);

    // Return the constructed segment tree
    return st;
}

// A recursive function to find the maximum rectangular area.
// It uses segment tree 'st' to find the minimum value in hist[l..r]
int getMaxAreaRec(int *hist, int *st, int n, int l, int r)
{
    // Base cases
    if (l > r) return INT_MIN;
    if (l == r) return hist[l];

    // Find index of the minimum value in given range
    // This takes O(Logn)time
    int m = RMQ(hist, st, n, l, r);

    /* Return maximum of following three possible cases
       a) Maximum area in Left of min value (not including the min)
       a) Maximum area in right of min value (not including the min)
       c) Maximum area including min */
    return max(getMaxAreaRec(hist, st, n, l, m-1),
               getMaxAreaRec(hist, st, n, m+1, r),
               (r-l+1)*(hist[m]) );
}
```

```
// The main function to find max area
int getMaxArea(int hist[], int n)
{
    // Build segment tree from given array. This takes
    // O(n) time
    int *st = constructST(hist, n);

    // Use recursive utility function to find the
    // maximum area
    return getMaxAreaRec(hist, st, n, 0, n-1);
}

// Driver program to test above functions
int main()
{
    int hist[] = {6, 1, 5, 4, 5, 2, 6};
    int n = sizeof(hist)/sizeof(hist[0]);
    cout << "Maximum area is " << getMaxArea(hist, n);
    return 0;
}
```

Output:

Maximum area is 12

This problem can be solved in linear time. See below [set 2](#) for linear time solution.
[Linear time solution for Largest Rectangular Area in a Histogram](#)

Source

<https://www.geeksforgeeks.org/largest-rectangular-area-in-a-histogram-set-1/>

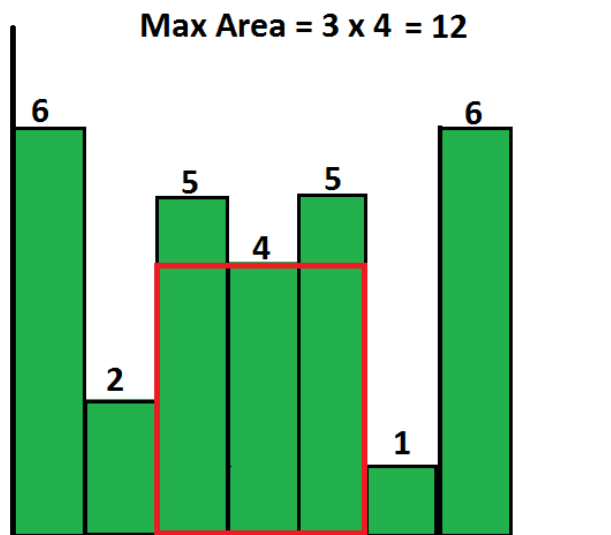
Chapter 62

Largest Rectangular Area in a Histogram | Set 2

Largest Rectangular Area in a Histogram | Set 2 - GeeksforGeeks

Find the largest rectangular area possible in a given histogram where the largest rectangle can be made of a number of contiguous bars. For simplicity, assume that all bars have same width and the width is 1 unit.

For example, consider the following histogram with 7 bars of heights {6, 2, 5, 4, 5, 1, 6}. The largest possible rectangle possible is 12 (see the below figure, the max area rectangle is highlighted in red)



We have discussed a [Divide and Conquer based \$O\(n \log n\)\$ solution](#) for this problem. In this

post, $O(n)$ time solution is discussed. Like the [previous post](#), width of all bars is assumed to be 1 for simplicity. For every bar 'x', we calculate the area with 'x' as the smallest bar in the rectangle. If we calculate such area for every bar 'x' and find the maximum of all areas, our task is done. How to calculate area with 'x' as smallest bar? We need to know index of the first smaller (smaller than 'x') bar on left of 'x' and index of first smaller bar on right of 'x'. Let us call these indexes as 'left index' and 'right index' respectively.

We traverse all bars from left to right, maintain a stack of bars. Every bar is pushed to stack once. A bar is popped from stack when a bar of smaller height is seen. When a bar is popped, we calculate the area with the popped bar as smallest bar. How do we get left and right indexes of the popped bar – the current index tells us the 'right index' and index of previous item in stack is the 'left index'. Following is the complete algorithm.

- 1) Create an empty stack.
- 2) Start from first bar, and do following for every bar 'hist[i]' where 'i' varies from 0 to n-1.
 -a) If stack is empty or hist[i] is higher than the bar at top of stack, then push 'i' to stack.
 -b) If this bar is smaller than the top of stack, then keep removing the top of stack while top of the stack is greater. Let the removed bar be hist[tp]. Calculate area of rectangle with hist[tp] as smallest bar. For hist[tp], the 'left index' is previous (previous to tp) item in stack and 'right index' is 'i' (current index).
- 3) If the stack is not empty, then one by one remove all bars from stack and do step 2.b for every removed bar.

Following is implementation of the above algorithm.

C++

```
// C++ program to find maximum rectangular area in
// linear time
#include<iostream>
#include<stack>
using namespace std;

// The main function to find the maximum rectangular
// area under given histogram with n bars
int getMaxArea(int hist[], int n)
{
    // Create an empty stack. The stack holds indexes
    // of hist[] array. The bars stored in stack are
    // always in increasing order of their heights.
    stack<int> s;

    int max_area = 0; // Initialize max area
    int tp; // To store top of stack
    int area_with_top; // To store area with top bar
                      // as the smallest bar

    // Run through all bars of given histogram
    int i = 0;
```

```
while (i < n)
{
    // If this bar is higher than the bar on top
    // stack, push it to stack
    if (s.empty() || hist[s.top()] <= hist[i])
        s.push(i++);

    // If this bar is lower than top of stack,
    // then calculate area of rectangle with stack
    // top as the smallest (or minimum height) bar.
    // 'i' is 'right index' for the top and element
    // before top in stack is 'left index'
    else
    {
        tp = s.top(); // store the top index
        s.pop(); // pop the top

        // Calculate the area with hist[tp] stack
        // as smallest bar
        area_with_top = hist[tp] * (s.empty() ? i :
                                   i - s.top() - 1);

        // update max area, if needed
        if (max_area < area_with_top)
            max_area = area_with_top;
    }
}

// Now pop the remaining bars from stack and calculate
// area with every popped bar as the smallest bar
while (s.empty() == false)
{
    tp = s.top();
    s.pop();
    area_with_top = hist[tp] * (s.empty() ? i :
                               i - s.top() - 1);

    if (max_area < area_with_top)
        max_area = area_with_top;
}

return max_area;
}

// Driver program to test above function
int main()
{
    int hist[] = {6, 2, 5, 4, 5, 1, 6};
```

```
int n = sizeof(hist)/sizeof(hist[0]);
cout << "Maximum area is " << getMaxArea(hist, n);
return 0;
}
```

Java

```
//Java program to find maximum rectangular area in linear time

import java.util.Stack;

public class RectArea
{
    // The main function to find the maximum rectangular area under given
    // histogram with n bars
    static int getMaxArea(int hist[], int n)
    {
        // Create an empty stack. The stack holds indexes of hist[] array
        // The bars stored in stack are always in increasing order of their
        // heights.
        Stack<Integer> s = new Stack<>();

        int max_area = 0; // Initialize max area
        int tp; // To store top of stack
        int area_with_top; // To store area with top bar as the smallest bar

        // Run through all bars of given histogram
        int i = 0;
        while (i < n)
        {
            // If this bar is higher than the bar on top stack, push it to stack
            if (s.empty() || hist[s.peek()] <= hist[i])
                s.push(i++);

            // If this bar is lower than top of stack, then calculate area of rectangle
            // with stack top as the smallest (or minimum height) bar. 'i' is
            // 'right index' for the top and element before top in stack is 'left index'
            else
            {
                tp = s.peek(); // store the top index
                s.pop(); // pop the top

                // Calculate the area with hist[tp] stack as smallest bar
                area_with_top = hist[tp] * (s.empty() ? i : i - s.peek() - 1);

                // update max area, if needed
                if (max_area < area_with_top)
                    max_area = area_with_top;
            }
        }
    }
}
```

```
    }
}

// Now pop the remaining bars from stack and calculate area with every
// popped bar as the smallest bar
while (s.empty() == false)
{
    tp = s.peek();
    s.pop();
    area_with_top = hist[tp] * (s.empty() ? i : i - s.peek() - 1);

    if (max_area < area_with_top)
        max_area = area_with_top;
}

return max_area;
}

// Driver program to test above function
public static void main(String[] args)
{
    int hist[] = { 6, 2, 5, 4, 5, 1, 6 };
    System.out.println("Maximum area is " + getMaxArea(hist, hist.length));
}
}
//This code is Contributed by Sumit Ghosh
```

Output:

Maximum area is 12

Time Complexity: Since every bar is pushed and popped only once, the time complexity of this method is $O(n)$.

References

<http://www.informatik.uni-ulm.de/acm/Locals/2003/html/histogram.html>

<http://www.informatik.uni-ulm.de/acm/Locals/2003/html/judge.html>

Thanks to [Ashish Anand](#) for suggesting initial solution. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/largest-rectangle-under-histogram/>

Chapter 63

Length of the longest valid substring

Length of the longest valid substring - GeeksforGeeks

Given a string consisting of opening and closing parenthesis, find length of the longest valid parenthesis substring.

Examples:

Input : ((()
Output : 2
Explanation : ()

Input:)()(()
Output : 4
Explanation: ()()

Input: ()((()))
Output: 6
Explanation: ()(())

A **Simple Approach** is to find all the substrings of given string. For every string, check if it is a valid string or not. If valid and length is more than maximum length so far, then update maximum length. We can check whether a substring is valid or not in linear time using a stack (See [this](#) for details). Time complexity of this solution is $O(n^2)$.

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.

Below are C++ and Python implementations of above algorithm.

C++

```
// C++ program to find length of the longest valid
// substring
#include<bits/stdc++.h>
using namespace std;

int findMaxLen(string str)
{
    int n = str.length();

    // Create a stack and push -1 as initial index to it.
    stack<int> stk;
    stk.push(-1);

    // Initialize result
    int result = 0;

    // Traverse all characters of given string
    for (int i=0; i<n; i++)
    {
        // If opening bracket, push index of it
        if (str[i] == '(')
            stk.push(i);
```

```
        else // If closing bracket, i.e., str[i] = ')'
        {
            // Pop the previous opening bracket's index
            stk.pop();

            // Check if this length formed with base of
            // current valid substring is more than max
            // so far
            if (!stk.empty())
                result = max(result, i - stk.top());

            // If stack is empty. push current index as
            // base for next valid substring (if any)
            else stk.push(i);
        }
    }

    return result;
}

// Driver program
int main()
{
    string str = "((()())";
    cout << findMaxLen(str) << endl;

    str = "()((()()))";
    cout << findMaxLen(str) << endl ;

    return 0;
}
```

Java

```
// Java program to find length of the longest valid
// substring

import java.util.Stack;

class Test
{
    // method to get length of the longest valid
    static int findMaxLen(String str)
    {
        int n = str.length();

        // Create a stack and push -1 as initial index to it.
        Stack<Integer> stk = new Stack<>();
```



```
stk.push(-1);

// Initialize result
int result = 0;

// Traverse all characters of given string
for (int i=0; i<n; i++)
{
    // If opening bracket, push index of it
    if (str.charAt(i) == '(')
        stk.push(i);

    else // If closing bracket, i.e., str[i] = ')'
    {
        // Pop the previous opening bracket's index
        stk.pop();

        // Check if this length formed with base of
        // current valid substring is more than max
        // so far
        if (!stk.empty())
            result = Math.max(result, i - stk.peek());

        // If stack is empty. push current index as
        // base for next valid substring (if any)
        else stk.push(i);
    }
}

return result;
}

// Driver method
public static void main(String[] args)
{
    String str = "(()())";
    System.out.println(findMaxLen(str));

    str = "()((()))";
    System.out.println(findMaxLen(str));
}
}
```

Python

```
# Python program to find length of the longest valid
# substring
```

```
def findMaxLen(string):
    n = len(string)

    # Create a stack and push -1 as initial index to it.
    stk = []
    stk.append(-1)

    # Initialize result
    result = 0

    # Traverse all characters of given string
    for i in xrange(n):

        # If opening bracket, push index of it
        if string[i] == '(':
            stk.append(i)

        else:    # If closing bracket, i.e., str[i] = ')'

            # Pop the previous opening bracket's index
            stk.pop()

            # Check if this length formed with base of
            # current valid substring is more than max
            # so far
            if len(stk) != 0:
                result = max(result, i - stk[len(stk)-1])

            # If stack is empty. push current index as
            # base for next valid substring (if any)
            else:
                stk.append(i)

    return result

# Driver program
string = "((()())"
print findMaxLen(string)

string = "()((())))"
print findMaxLen(string)

# This code is contributed by Bhavya Jain
```

Output:

6

Explanation with example:

Input: str = "()()"

Initialize result as 0 and stack with one item -1.

For i = 0, str[0] = '(', we push 0 in stack

For i = 1, str[1] = '(', we push 1 in stack

For i = 2, str[2] = ')', currently stack has [-1, 0, 1], we pop from the stack and the stack now is [-1, 0] and length of current valid substring becomes 2 (we get this 2 by subtracting stack top from current index).
Since current length is more than current result, we update result.

For i = 3, str[3] = '(', we push again, stack is [-1, 0, 3].

For i = 4, str[4] = ')', we pop from the stack, stack becomes [-1, 0] and length of current valid substring becomes 4 (we get this 4 by subtracting stack top from current index).
Since current length is more than current result, we update result.

Thanks to Gaurav Ahirwar and [Ekta Goel](#). for suggesting above approach.

Source

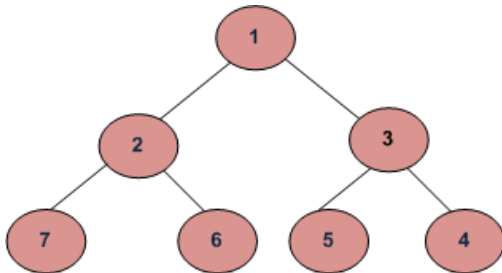
<https://www.geeksforgeeks.org/length-of-the-longest-valid-substring/>

Chapter 64

Level order traversal in spiral form

Level order traversal in spiral form - GeeksforGeeks

Write a function to print spiral order traversal of a tree. For below tree, function should print 1, 2, 3, 4, 5, 6, 7.



Method 1 (Recursive)

This problem can be seen as an extension of the [level order traversal](#) post.

To print the nodes in spiral order, nodes at different levels should be printed in alternating order. An additional Boolean variable *ltr* is used to change printing order of levels. If *ltr* is 1 then `printGivenLevel()` prints nodes from left to right else from right to left. Value of *ltr* is flipped in each iteration to change the order.

Function to print level order traversal of tree

```
printSpiral(tree)
    bool ltr = 0;
    for d = 1 to height(tree)
        printGivenLevel(tree, d, ltr);
        ltr ~= ltr /*flip ltr*/
```

Function to print all nodes at a given level

```
printGivenLevel(tree, level, ltr)
if tree is NULL then return;
if level is 1, then
    print(tree->data);
else if level greater than 1, then
    if(ltr)
        printGivenLevel(tree->left, level-1, ltr);
        printGivenLevel(tree->right, level-1, ltr);
    else
        printGivenLevel(tree->right, level-1, ltr);
        printGivenLevel(tree->left, level-1, ltr);
```

Following is C implementation of above algorithm.

C

```
// C program for recursive level order traversal in spiral form
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Function prototypes */
void printGivenLevel(struct node* root, int level, int ltr);
int height(struct node* node);
struct node* newNode(int data);

/* Function to print spiral traversal of a tree*/
void printSpiral(struct node* root)
{
    int h = height(root);
    int i;

    /*ltr -> Left to Right. If this variable is set,
       then the given level is traversed from left to right. */
    bool ltr = false;
    for(i=1; i<=h; i++)
    {
```

```
    printGivenLevel(root, i, ltr);

    /*Revert ltr to traverse next level in opposite order*/
    ltr = !ltr;
}
}

/* Print nodes at a given level */
void printGivenLevel(struct node* root, int level, int ltr)
{
    if(root == NULL)
        return;
    if(level == 1)
        printf("%d ", root->data);
    else if (level > 1)
    {
        if(ltr)
        {
            printGivenLevel(root->left, level-1, ltr);
            printGivenLevel(root->right, level-1, ltr);
        }
        else
        {
            printGivenLevel(root->right, level-1, ltr);
            printGivenLevel(root->left, level-1, ltr);
        }
    }
}

/* Compute the "height" of a tree -- the number of
   nodes along the longest path from the root node
   down to the farthest leaf node.*/
int height(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lheight = height(node->left);
        int rheight = height(node->right);

        /* use the larger one */
        if (lheight > rheight)
            return(lheight+1);
        else return(rheight+1);
    }
}
```

```
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));

    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left = newNode(7);
    root->left->right = newNode(6);
    root->right->left = newNode(5);
    root->right->right = newNode(4);
    printf("Spiral Order traversal of binary tree is \n");
    printSpiral(root);

    return 0;
}
```

Java

```
// Java program for recursive level order traversal in spiral form

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    public Node(int d)
    {
        data = d;
        left = right = null;
    }
}
```

```
class BinaryTree
{
    Node root;

    // Function to print the spiral traversal of tree
    void printSpiral(Node node)
    {
        int h = height(node);
        int i;

        /* ltr -> left to right. If this variable is set then the
           given label is transversed from left to right */
        boolean ltr = false;
        for (i = 1; i <= h; i++)
        {
            printGivenLevel(node, i, ltr);

            /*Revert ltr to traverse next level in opposite order*/
            ltr = !ltr;
        }
    }

    /* Compute the "height" of a tree -- the number of
       nodes along the longest path from the root node
       down to the farthest leaf node.*/
    int height(Node node)
    {
        if (node == null)
            return 0;
        else
        {
            /* compute the height of each subtree */
            int lheight = height(node.left);
            int rheight = height(node.right);

            /* use the larger one */
            if (lheight > rheight)
                return (lheight + 1);
            else
                return (rheight + 1);
        }
    }

    /* Print nodes at a given level */
    void printGivenLevel(Node node, int level, boolean ltr)
    {
```



```

        if (node == null)
            return;
        if (level == 1)
            System.out.print(node.data + " ");
        else if (level > 1)
        {
            if (ltr != false)
            {
                printGivenLevel(node.left, level - 1, ltr);
                printGivenLevel(node.right, level - 1, ltr);
            }
            else
            {
                printGivenLevel(node.right, level - 1, ltr);
                printGivenLevel(node.left, level - 1, ltr);
            }
        }
    }
}
/* Driver program to test the above functions */
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(7);
    tree.root.left.right = new Node(6);
    tree.root.right.left = new Node(5);
    tree.root.right.right = new Node(4);
    System.out.println("Spiral order traversal of Binary Tree is ");
    tree.printSpiral(tree.root);
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)

```

Output:

```

Spiral Order traversal of binary tree is
1 2 3 4 5 6 7

```

Time Complexity: Worst case time complexity of the above method is $O(n^2)$. Worst case occurs in case of skewed trees.

Method 2 (Iterative)

We can print spiral order traversal in $O(n)$ time and $O(n)$ extra space. The idea is to use two stacks. We can use one stack for printing from left to right and other stack for printing from right to left. In every iteration, we have nodes of one level in one of the stacks. We

print the nodes, and push nodes of next level in other stack.

C++

```
// C++ implementation of a O(n) time method for spiral order traversal
#include <iostream>
#include <stack>
using namespace std;

// Binary Tree node
struct node
{
    int data;
    struct node *left, *right;
};

void printSpiral(struct node *root)
{
    if (root == NULL) return;    // NULL check

    // Create two stacks to store alternate levels
    stack<struct node*> s1; // For levels to be printed from right to left
    stack<struct node*> s2; // For levels to be printed from left to right

    // Push first level to first stack 's1'
    s1.push(root);

    // Keep printing while any of the stacks has some nodes
    while (!s1.empty() || !s2.empty())
    {
        // Print nodes of current level from s1 and push nodes of
        // next level to s2
        while (!s1.empty())
        {
            struct node *temp = s1.top();
            s1.pop();
            cout << temp->data << " ";

            // Note that is right is pushed before left
            if (temp->right)
                s2.push(temp->right);
            if (temp->left)
                s2.push(temp->left);
        }

        // Print nodes of current level from s2 and push nodes of
        // next level to s1
        while (!s2.empty())
```

```
{
    struct node *temp = s2.top();
    s2.pop();
    cout << temp->data << " ";

    // Note that is left is pushed before right
    if (temp->left)
        s1.push(temp->left);
    if (temp->right)
        s1.push(temp->right);
}
}

// A utility function to create a new node
struct node* newNode(int data)
{
    struct node* node = new struct node;
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

int main()
{
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left = newNode(7);
    root->left->right = newNode(6);
    root->right->left = newNode(5);
    root->right->right = newNode(4);
    cout << "Spiral Order traversal of binary tree is \n";
    printSpiral(root);

    return 0;
}
```

Java

```
// Java implementation of an O(n) approach of level order
// traversal in spiral form

import java.util.*;

// A Binary Tree node
```

```
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    static Node root;

    void printSpiral(Node node)
    {
        if (node == null)
            return;    // NULL check

        // Create two stacks to store alternate levels
        Stack<Node> s1 = new Stack<Node>(); // For levels to be printed from right to left
        Stack<Node> s2 = new Stack<Node>(); // For levels to be printed from left to right

        // Push first level to first stack 's1'
        s1.push(node);

        // Keep printing while any of the stacks has some nodes
        while (!s1.empty() || !s2.empty())
        {
            // Print nodes of current level from s1 and push nodes of
            // next level to s2
            while (!s1.empty())
            {
                Node temp = s1.peek();
                s1.pop();
                System.out.print(temp.data + " ");

                // Note that right is pushed before left
                if (temp.right != null)
                    s2.push(temp.right);

                if (temp.left != null)
                    s2.push(temp.left);
            }
        }
    }
}
```

```
// Print nodes of current level from s2 and push nodes of
// next level to s1
while (!s2.empty())
{
    Node temp = s2.peek();
    s2.pop();
    System.out.print(temp.data + " ");

    // Note that left is pushed before right
    if (temp.left != null)
        s1.push(temp.left);
    if (temp.right != null)
        s1.push(temp.right);
}
}

public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(7);
    tree.root.left.right = new Node(6);
    tree.root.right.left = new Node(5);
    tree.root.right.right = new Node(4);
    System.out.println("Spiral Order traversal of Binary Tree is ");
    tree.printSpiral(root);
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output:

```
Spiral Order traversal of binary tree is
1 2 3 4 5 6 7
```

Please write comments if you find any bug in the above program/algorithm; or if you want to share more information about spiral traversal.

Source

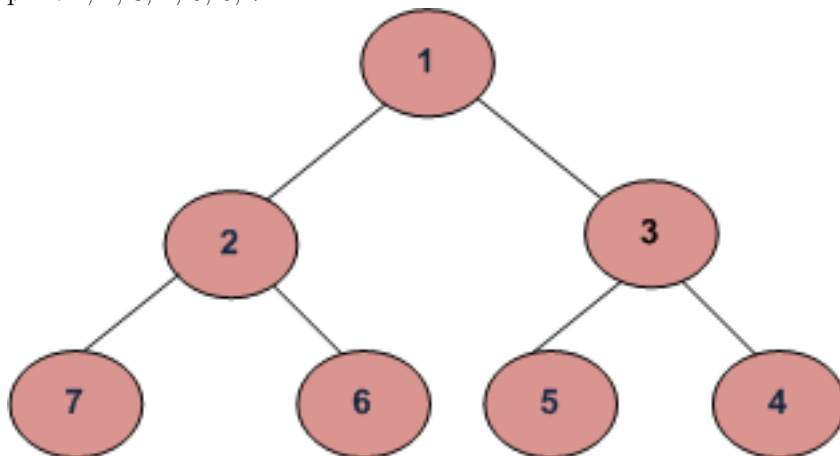
<https://www.geeksforgeeks.org/level-order-traversal-in-spiral-form/>

Chapter 65

Level order traversal in spiral form | Using one stack and one queue

Level order traversal in spiral form | Using one stack and one queue - GeeksforGeeks

Write a function to print spiral order traversal of a tree. For below tree, function should print 1, 2, 3, 4, 5, 6, 7.



You are allowed to use only one stack.

We have seen [recursive and iterative solutions using two stacks](#). In this post, a solution with one stack and one queue is discussed. The idea is to keep on entering nodes like normal level order traversal, but during printing, in alternative turns push them onto the stack and print them, and in other traversals, just print them the way they are present in the queue.

Following is the CPP implementation of the idea.

```
// CPP program to print level order traversal
```

```
// in spiral form using one queue and one stack.
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node *left, *right;
};

/* Utility function to create a new tree node */
Node* newNode(int val)
{
    Node* new_node = new Node;
    new_node->data = val;
    new_node->left = new_node->right = NULL;
    return new_node;
}

/* Function to print a tree in spiral form
   using one stack */
void printSpiralUsingOneStack(Node* root)
{
    if (root == NULL)
        return;

    stack<int> s;
    queue<Node*> q;

    bool reverse = true;
    q.push(root);
    while (!q.empty()) {

        int size = q.size();
        while (size) {
            Node* p = q.front();
            q.pop();

            // if reverse is true, push node's
            // data onto the stack, else print it
            if (reverse)
                s.push(p->data);
            else
                cout << p->data << " ";

            if (p->left)
                q.push(p->left);
            if (p->right)
                q.push(p->right);
        }

        if (reverse) {
            while (!s.empty())
                cout << s.top() << " ";
            s.empty();
        }
        reverse = !reverse;
    }
}
```

```
        size--;  
    }  
  
    // print nodes from the stack if  
    // reverse is true  
    if (reverse) {  
        while (!s.empty()) {  
            cout << s.top() << " ";  
            s.pop();  
        }  
    }  
  
    // the next row has to be printed as  
    // it is, hence change the value of  
    // reverse  
    reverse = !reverse;  
}  
}  
  
/*Driver program to test the above functions*/  
int main()  
{  
    Node* root = newNode(1);  
    root->left = newNode(2);  
    root->right = newNode(3);  
    root->left->left = newNode(7);  
    root->left->right = newNode(6);  
    root->right->left = newNode(5);  
    root->right->right = newNode(4);  
    printSpiralUsingOneStack(root);  
    return 0;  
}
```

Output:

1 2 3 4 5 6 7

Time Complexity : $O(n)$

Auxiliary Space : $O(n)$

Source

<https://www.geeksforgeeks.org/level-order-traversal-in-spiral-form-using-one-stack-and-one-queue/>

Chapter 66

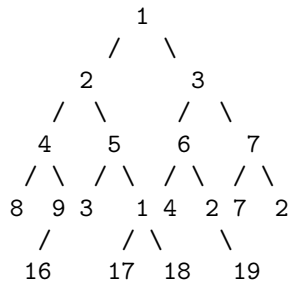
Level order traversal with direction change after every two levels

Level order traversal with direction change after every two levels - GeeksforGeeks

Given a binary tree, print the level order traversal in such a way that first two levels are printed from left to right, next two levels are printed from right to left, then next two from left to right and so on. So, the problem is to reverse the direction of level order traversal of binary tree after every two levels.

Examples:

Input:



Output:

```
1
2 3
7 6 5 4
2 7 2 4 1 3 9 8
16 17 18 19
```

In the above example, first two levels are printed from left to right, next two

levels are printed from right to left,
and then last level is printed from
left to right.

Approach:

We make use of queue and stack here. Queue is used for performing normal level order traversal. Stack is used for reversing the direction of traversal after every two levels.

While doing normal level order traversal, first two levels nodes are printed at the time when they are popped out from the queue. For the next two levels, we instead of printing the nodes, pushed them onto the stack. When all nodes of current level are popped out, we print the nodes in the stack. In this way, we print the nodes in right to left order by making use of the stack. Now for the next two levels we again do normal level order traversal for printing nodes from left to right. Then for the next two nodes, we make use of the stack for achieving right to left order.

In this way, we will achieve desired modified level order traversal by making use of queue and stack.

```
// CPP program to print Zig-Zag traversal
// in groups of size 2.
#include <iostream>
#include <queue>
#include <stack>
using namespace std;

// A Binary Tree Node
struct Node {
    struct Node* left;
    int data;
    struct Node* right;
};

/* Function to print the level order of
given binary tree. Direction of printing
level order traversal of binary tree changes
after every two levels */
void modifiedLevelOrder(struct Node* node)
{
    // For null root
    if (node == NULL)
        return;

    if (node->left == NULL && node->right == NULL) {
        cout << node->data;
        return;
    }

    // Maintain a queue for normal level order traversal
    queue<Node*> myQueue;
```

```
/* Maintain a stack for printing nodes in reverse
   order after they are popped out from queue.*/
stack<Node*> myStack;

struct Node* temp = NULL;

// sz is used for storing the count of nodes in a level
int sz;

// Used for changing the direction of level order traversal
int ct = 0;

// Used for changing the direction of level order traversal
bool rightToLeft = false;

// Push root node to the queue
myQueue.push(node);

// Run this while loop till queue got empty
while (!myQueue.empty()) {
    ct++;

    sz = myQueue.size();

    // Do a normal level order traversal
    for (int i = 0; i < sz; i++) {
        temp = myQueue.front();
        myQueue.pop();

        /*For printing nodes from left to right,
        simply print the nodes in the order in which
        they are being popped out from the queue.*/
        if (rightToLeft == false)
            cout << temp->data << " ";

        /* For printing nodes from right to left,
        push the nodes to stack instead of printing them.*/
        else
            myStack.push(temp);

        if (temp->left)
            myQueue.push(temp->left);

        if (temp->right)
            myQueue.push(temp->right);
    }
}
```

```

        if (rightToLeft == true) {

            // for printing the nodes in order
            // from right to left
            while (!myStack.empty()) {
                temp = myStack.top();
                myStack.pop();

                cout << temp->data << " ";
            }

            /*Change the direction of printing
            nodes after every two levels.*/
            if (ct == 2) {
                rightToLeft = !rightToLeft;
                ct = 0;
            }

            cout << "\n";
        }
    }

    // Utility function to create a new tree node
    Node* newNode(int data)
    {
        Node* temp = new Node;
        temp->data = data;
        temp->left = temp->right = NULL;
        return temp;
    }

    // Driver program to test above functions
    int main()
    {
        // Let us create binary tree
        Node* root = newNode(1);
        root->left = newNode(2);
        root->right = newNode(3);
        root->left->left = newNode(4);
        root->left->right = newNode(5);
        root->right->left = newNode(6);
        root->right->right = newNode(7);
        root->left->left->left = newNode(8);
        root->left->left->right = newNode(9);
        root->left->right->left = newNode(3);
        root->left->right->right = newNode(1);
        root->right->left->left = newNode(4);
    }

```

```
    root->right->left->right = newNode(2);
    root->right->right->left = newNode(7);
    root->right->right->right = newNode(2);
    root->left->right->left->left = newNode(16);
    root->left->right->left->right = newNode(17);
    root->right->left->right->left = newNode(18);
    root->right->right->left->right = newNode(19);

    modifiedLevelOrder(root);

    return 0;
}
```

Output:

```
1
2 3
7 6 5 4
2 7 2 4 1 3 9 8
16 17 18 19
```

Time Complexity: Each node is traversed at most twice while doing level order traversal, so time complexity would be $O(n)$.

Approach 2:

We make use of queue and stack here, but in a different way. Using macros `#define ChangeDirection(Dir) ((Dir) = 1 - (Dir))`. In following implementation directs the order of push operations in both queue or stack.

In this way, we will achieve desired modified level order traversal by making use of queue and stack.

```
// C++ program to print Zig-Zag traversal
// in groups of size 2.
#include <iostream>
#include <stack>
#include <queue>

using namespace std;

#define LEFT 0
#define RIGHT 1
#define ChangeDirection(Dir) ((Dir) = 1 - (Dir))

// A Binary Tree Node
struct node
{
    int data;
```

```
    struct node *left, *right;
};

// Utility function to create a new tree node
node* newNode(int data)
{
    node* temp = new node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

/* Function to print the level order of
   given binary tree. Direction of printing
   level order traversal of binary tree changes
   after every two levels */
void modifiedLevelOrder(struct node *root)
{
    if (!root)
        return ;

    int dir = LEFT;
    struct node *temp;
    queue <struct node *> Q;
    stack <struct node *> S;

    S.push(root);

    // Run this while loop till queue got empty
    while (!Q.empty() || !S.empty())
    {
        while (!S.empty())
        {
            temp = S.top();
            S.pop();
            cout << temp->data << " ";

            if (dir == LEFT) {
                if (temp->left)
                    Q.push(temp->left);
                if (temp->right)
                    Q.push(temp->right);
            }
            /* For printing nodes from right to left,
               push the nodes to stack instead of printing them.*/
            else {
                if (temp->right)
                    Q.push(temp->right);
            }
        }
    }
}
```

```
        if (temp->left)
            Q.push(temp->left);
    }
}

cout << endl;

    // for printing the nodes in order
    // from right to left
while (!Q.empty())
{
    temp = Q.front();
    Q.pop();
    cout << temp->data << " ";

    if (dir == LEFT) {
        if (temp->left)
            S.push(temp->left);
        if (temp->right)
            S.push(temp->right);
    } else {
        if (temp->right)
            S.push(temp->right);
        if (temp->left)
            S.push(temp->left);
    }
}
cout << endl;

    // Change the direction of traversal.
    ChangeDirection(dir);
}

}

// Driver program to test above functions
int main()
{
    // Let us create binary tree
    node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->left->left->left = newNode(8);
    root->left->left->right = newNode(9);
    root->left->right->left = newNode(3);
}
```

```
root->left->right->right = newNode(1);
root->right->left->left = newNode(4);
root->right->left->right = newNode(2);
root->right->right->left = newNode(7);
root->right->right->right = newNode(2);
root->left->right->left->left = newNode(16);
root->left->right->left->right = newNode(17);
root->right->left->right->left = newNode(18);
root->right->right->left->right = newNode(19);

modifiedLevelOrder(root);

return 0;
}
```

Output:

```
1
2 3
7 6 5 4
2 7 2 4 1 3 9 8
16 17 18 19
```

Time Complexity: every node is also traversed twice. There time complexity is still $O(n)$.

Improved By : [vhinf2047](#)

Source

<https://www.geeksforgeeks.org/level-order-traversal-direction-change-every-two-levels/>

Chapter 67

Maximum length of rod for Q-th person

Maximum length of rod for Q-th person - GeeksforGeeks

Given lengths of n rods in an array $a[]$. If any person picks any rod, half of the longest rod (or $(\max + 1) / 2$) is assigned and remaining part $(\max - 1) / 2$ is put back. It may be assumed that sufficient number of rods are always available, answer M queries given in an array $q[]$ to find the largest length of rod available for q^{ith} person, provided q^i is a valid person number starting from 1.

Examples :

Input : $a[] = \{6, 5, 9, 10, 12\}$
 $q[] = \{1, 3\}$

Output : 12 9

The first person gets maximum length as 12.

We remove 12 from array and put back $(12 - 1) / 2 = 5$.

Second person gets maximum length as 10.

We put back $(10 - 1) / 2$ which is 4.

Third person gets maximum length as 9.

Input : $a[] = \{6, 5, 9, 10, 12\}$
 $q[] = \{3, 1, 2, 7, 4, 8, 9, 5, 10, 6\}$

Output : 9 12 10 5 6 4 3 6 3 5

Approach :

Use a stack and a queue. First sort all the lengths and push them onto a stack. Now, take the top element of stack, and divide by 2 and push the remaining length to queue. Now, from next customer onwards :

1. If stack is empty, pop front queue and push back to queue. It's half (front / 2), if non zero.

2. If queue is empty, pop from stack and push to queue it's half ($\text{top} / 2$), if non zero.
3. If both are non empty, compare top and front, which ever is larger should be popped, divided by 2 and then pushed back.
4. If both are empty, store is empty! Stop here!

At each step above store the length available to i^{th} customer in separate array, say "ans". Now, start answering the queries by giving $\text{ans}[Q_i]$ as output.

Below is the implementation of above approach :

```
// CPP code to find the length of largest
// rod available for Q-th customer
#include <bits/stdc++.h>
using namespace std;

// function to find largest length of
// rod available for Q-th customer
vector<int> maxRodLength(int ar[],
                        int n, int m)
{
    queue<int> q;

    // sort the rods according to lengths
    sort(ar, ar + n);

    // Push sorted elements to a stack
    stack<int> s;
    for (int i = 0; i < n; i++)
        s.push(ar[i]);

    vector<int> ans;

    while (!s.empty() || !q.empty()) {
        int val;

        // If queue is empty -> pop from stack
        // and push to queue it's half(top/2),
        // if non zero.
        if (q.empty()) {
            val = s.top();
            ans.push_back(val);
            s.pop();
            val /= 2;

            if (val)
                q.push(val);
        }
        // If stack is empty -> pop front from
```

```
// queue and push back to queue it's
// half(front/2), if non zero.
else if (s.empty()) {
    val = q.front();
    ans.push_back(val);
    q.pop();
    val /= 2;
    if (val != 0)
        q.push(val);
}
// If both are non empty ->
// compare top and front, whichever is
// larger should be popped, divided by 2
// and then pushed back.
else {
    val = s.top();
    int fr = q.front();
    if (fr > val) {
        ans.push_back(fr);
        q.pop();
        fr /= 2;
        if (fr)
            q.push(fr);
    }
    else {
        ans.push_back(val);
        s.pop();
        val /= 2;
        if (val)
            q.push(val);
    }
}
}

return ans;
}

// Driver code
int main()
{
    // n : number of rods
    // m : number of queries
    int n = 5, m = 10;

    int ar[n] = { 6, 5, 9, 10, 12 };

    vector<int> ans = maxRodLength(ar, n, m);
```

```
int query[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int size = sizeof(query) / sizeof(query[0]);
for (int i = 0; i < size; i++)
    cout << ans[query[i] - 1] << " ";

return 0;
}
```

Output:

12 10 9 6 6 5 5 4 3 3

Time complexity : $O(N \log(N))$

Source

<https://www.geeksforgeeks.org/maximum-length-rod-q-th-person/>

Chapter 68

Maximum product of indexes of next greater on left and right

Maximum product of indexes of next greater on left and right - GeeksforGeeks

Given an array $a[1..N]$. For each element at position i ($1 \leq i \leq N$). Where

1. $L(i)$ is defined as closest index j such that $j < i$ and $a[j] > a[i]$. If no such j exists then $L(i) = 0$.
2. $R(i)$ is defined as closest index k such that $k > i$ and $a[k] > a[i]$. If no such k exists then $R(i) = 0$.

$LRProduct(i) = L(i) * R(i)$.

We need to find an index with maximum LRProduct

Examples:

Input : 1 1 1 1 0 1 1 1 1 1

Output : 24

For $\{1, 1, 1, 1, 0, 1, 1, 1, 1, 1\}$ all element are same except 0. So only for zero their exist greater element and for others it will be zero. for zero, on left 4th element is closest and greater than zero and on right 6th element is closest and greater. so maximum product will be $4 * 6 = 24$.

Input : 5 4 3 4 5

Output : 8

For $\{5, 4, 3, 4, 5\}$, $L[] = \{0, 1, 2, 1, 0\}$ and $R[] = \{0, 5, 4, 5, 0\}$,

$LRProduct = \{0, 5, 8, 5, 0\}$ and max in this is 8.

Note: Taking starting index as 1 for finding LRproduct.

This problem is based on [Next Greater Element](#).

From the current position, we need to find the closest greater element on its left and right side.

So to find next greater element, we used stack one from left and one from right. simply we are checking which element is greater and storing their index at specified position.

1- if stack is empty, push current index.

2- if stack is not empty

....a) if current element is greater than top element then store the index of current element on index of top element.

Do this, once traversing array element from left and once from right and form the left and right array, then, multiply them to find max product value.

C++

```
// C++ program to find the max
// LRproduct[i] among all i
#include <bits/stdc++.h>
using namespace std;
#define MAX 1000

// function to find just next greater
// element in left side
vector<int> nextGreaterInLeft(int a[], int n)
{
    vector<int> left_index(MAX, 0);
    stack<int> s;

    for (int i = n - 1; i >= 0; i--) {

        // checking if current element is greater than top
        while (!s.empty() && a[i] > a[s.top() - 1]) {
            int r = s.top();
            s.pop();

            // on index of top store the current element
            // index which is just greater than top element
            left_index[r - 1] = i + 1;
        }

        // else push the current element in stack
        s.push(i + 1);
    }
    return left_index;
}

// function to find just next greater element
```

```
// in right side
vector<int> nextGreaterInRight(int a[], int n)
{
    vector<int> right_index(MAX, 0);
    stack<int> s;
    for (int i = 0; i < n; ++i) {

        // checking if current element is greater than top
        while (!s.empty() && a[i] > a[s.top() - 1]) {
            int r = s.top();
            s.pop();

            // on index of top store the current element
            // index which is just greater than top element
            // stored index should be start with 1
            right_index[r - 1] = i + 1;
        }

        // else push the current element in stack
        s.push(i + 1);
    }
    return right_index;
}

// Function to find maximum LR product
int LRProduct(int arr[], int n)
{
    // for each element storing the index of just
    // greater element in left side
    vector<int> left = nextGreaterInLeft(arr, n);

    // for each element storing the index of just
    // greater element in right side
    vector<int> right = nextGreaterInRight(arr, n);
    int ans = -1;
    for (int i = 1; i <= n; i++) {

        // finding the max index product
        ans = max(ans, left[i] * right[i]);
    }

    return ans;
}

// Drivers code
int main()
{
    int arr[] = { 5, 4, 3, 4, 5 };
```

```
int n = sizeof(arr) / sizeof(arr[1]);

cout << LRProduct(arr, n);

return 0;
}
```

Java

```
// Java program to find the
// max LRproduct[i] among all i
import java.io.*;
import java.util.*;

class GFG
{
    static int MAX = 1000;

    // function to find just next
    // greater element in left side
    static int[] nextGreaterInLeft(int []a,
                                    int n)
    {
        int []left_index = new int[MAX];
        Stack<Integer> s = new Stack<Integer>();

        for (int i = n - 1; i >= 0; i--)
        {
            // checking if current
            // element is greater than top
            while (s.size() != 0 &&
                    a[i] > a[s.peek() - 1])
            {
                int r = s.peek();
                s.pop();

                // on index of top store
                // the current element
                // index which is just
                // greater than top element
                left_index[r - 1] = i + 1;
            }

            // else push the current
            // element in stack
            s.push(i + 1);
        }
    }
}
```



```
        return left_index;
    }

    // function to find just next
    // greater element in right side
    static int[] nextGreaterInRight(int []a,
                                    int n)
    {
        int []right_index = new int[MAX];
        Stack<Integer> s = new Stack<Integer>();
        for (int i = 0; i < n; ++i) {

            // checking if current element
            // is greater than top
            while (s.size() != 0 &&
                   a[i] > a[s.peek() - 1])
            {
                int r = s.peek();
                s.pop();

                // on index of top store
                // the current element index
                // which is just greater than
                // top element stored index
                // should be start with 1
                right_index[r - 1] = i + 1;
            }

            // else push the current
            // element in stack
            s.push(i + 1);
        }
        return right_index;
    }

    // Function to find
    // maximum LR product
    static int LRProduct(int []arr, int n)
    {
        // for each element storing
        // the index of just greater
        // element in left side
        int []left = nextGreaterInLeft(arr, n);

        // for each element storing
        // the index of just greater
        // element in right side
```

```
int []right = nextGreaterInRight(arr, n);
int ans = -1;
for (int i = 1; i <= n; i++)
{

    // finding the max
    // index product
    ans = Math.max(ans, left[i] *
                    right[i]);
}

return ans;
}

// Driver code
public static void main(String args[])
{
    int []arr = new int[]{ 5, 4, 3, 4, 5 };
    int n = arr.length;

    System.out.print(LRProduct(arr, n));
}

// This code is contributed by
// Manish Shaw(manishshaw1)
```

C#

```
// C# program to find the max LRproduct[i]
// among all i
using System;
using System.Collections.Generic;

class GFG {

    static int MAX = 1000;

    // function to find just next greater
    // element in left side
    static int[] nextGreaterInLeft(int []a, int n)
    {
        int []left_index = new int[MAX];
        Stack<int> s = new Stack<int>();

        for (int i = n - 1; i >= 0; i--) {

            // checking if current element is
```

```
// greater than top
while (s.Count != 0 && a[i] > a[s.Peek() - 1])
{
    int r = s.Peek();
    s.Pop();

    // on index of top store the current
    // element index which is just greater
    // than top element
    left_index[r - 1] = i + 1;
}

// else push the current element in stack
s.Push(i + 1);
}
return left_index;
}

// function to find just next greater element
// in right side
static int[] nextGreaterInRight(int []a, int n)
{
    int []right_index = new int[MAX];
    Stack<int> s = new Stack<int>();
    for (int i = 0; i < n; ++i) {

        // checking if current element is
        // greater than top
        while (s.Count != 0 && a[i] > a[s.Peek() - 1])
        {
            int r = s.Peek();
            s.Pop();

            // on index of top store the current
            // element index which is just greater
            // than top element stored index should
            // be start with 1
            right_index[r - 1] = i + 1;
        }

        // else push the current element in stack
        s.Push(i + 1);
    }
    return right_index;
}

// Function to find maximum LR product
static int LRProduct(int []arr, int n)
```

```
{

    // for each element storing the index of just
    // greater element in left side
    int []left = nextGreaterInLeft(arr, n);

    // for each element storing the index of just
    // greater element in right side
    int []right = nextGreaterInRight(arr, n);
    int ans = -1;
    for (int i = 1; i <= n; i++) {

        // finding the max index product
        ans = Math.Max(ans, left[i] * right[i]);
    }

    return ans;
}

// Drivers code
static void Main()
{
    int []arr = new int[] { 5, 4, 3, 4, 5 };
    int n = arr.Length;

    Console.Write(LRProduct(arr, n));
}

// This code is contributed by Manish Shaw
// (manishshaw1)
```

Output:

8

Improved By : [manishshaw1](#), [Vivek Agarwal](#)

Source

<https://www.geeksforgeeks.org/maximum-product-of-indexes-of-next-greater-on-left-and-right/>

Chapter 69

Maximum size rectangle binary sub-matrix with all 1s

Maximum size rectangle binary sub-matrix with all 1s - GeeksforGeeks

Given a binary matrix, find the maximum size rectangle binary-sub-matrix with all 1's.

```
Input :   0 1 1 0
          1 1 1 1
          1 1 1 1
          1 1 0 0
```

```
Output :  1 1 1 1
          1 1 1 1
```

We have discussed a [dynamic programming based solution for finding largest square with 1s](#).

In this post an interesting method is discussed that uses [largest rectangle under histogram](#) as a subroutine. Below are steps. The idea is to update each column of a given row with corresponding column of previous row and find largest histogram area for for that row.

Step 1: Find maximum area for row[0]

Step 2:

```
for each row in 1 to N - 1
    for each column in that row
        if A[row][column] == 1
            update A[row][column] with
                A[row][column] += A[row - 1][column]
    find area for that row
    and update maximum area so far
```

Illustration :

```
step 1:    0 1 1 0  maximum area  = 2
step 2:
  row 1    1 2 2 1  area = 4, maximum area becomes 4
  row 2    2 3 3 2  area = 8, maximum area becomes 8
  row 3    3 4 0 0  area = 6, maximum area remains 8
```

Below is the implementation. It is strongly recommended to refer [this](#) post first as most of the code taken from there.

C++

```
// C++ program to find largest rectangle with all 1s
// in a binary matrix
#include<bits/stdc++.h>
using namespace std;

// Rows and columns in input matrix
#define R 4
#define C 4

// Finds the maximum area under the histogram represented
// by histogram. See below article for details.
// https://www.geeksforgeeks.org/largest-rectangle-under-histogram/
int maxHist(int row[])
{
    // Create an empty stack. The stack holds indexes of
    // hist[] array/ The bars stored in stack are always
    // in increasing order of their heights.
    stack<int> result;

    int top_val;    // Top of stack

    int max_area = 0; // Initialize max area in current
                     // row (or histogram)

    int area = 0;    // Initialize area with current top

    // Run through all bars of given histogram (or row)
    int i = 0;
    while (i < C)
    {
        // If this bar is higher than the bar on top stack,
        // push it to stack
        if (result.empty() || row[result.top()] <= row[i])
```

```
        result.push(i++);

    else
    {
        // If this bar is lower than top of stack, then
        // calculate area of rectangle with stack top as
        // the smallest (or minimum height) bar. 'i' is
        // 'right index' for the top and element before
        // top in stack is 'left index'
        top_val = row[result.top()];
        result.pop();
        area = top_val * i;

        if (!result.empty())
            area = top_val * (i - result.top() - 1);
        max_area = max(area, max_area);
    }
}

// Now pop the remaining bars from stack and calculate area
// with every popped bar as the smallest bar
while (!result.empty())
{
    top_val = row[result.top()];
    result.pop();
    area = top_val * i;
    if (!result.empty())
        area = top_val * (i - result.top() - 1);

    max_area = max(area, max_area);
}
return max_area;
}

// Returns area of the largest rectangle with all 1s in A[][]
int maxRectangle(int A[][C])
{
    // Calculate area for first row and initialize it as
    // result
    int result = maxHist(A[0]);

    // iterate over row to find maximum rectangular area
    // considering each row as histogram
    for (int i = 1; i < R; i++)
    {
        for (int j = 0; j < C; j++)
```

```
        // if A[i][j] is 1 then add A[i -1][j]
        if (A[i][j]) A[i][j] += A[i - 1][j];

        // Update result if area with current row (as last row)
        // of rectangle) is more
        result = max(result, maxHist(A[i]));
    }

    return result;
}

// Driver code
int main()
{
    int A[][C] = { {0, 1, 1, 0},
                    {1, 1, 1, 1},
                    {1, 1, 1, 1},
                    {1, 1, 0, 0},
                    };

    cout << "Area of maximum rectangle is "
          << maxRectangle(A);

    return 0;
}
```

Java

```
// Java program to find largest rectangle with all 1s
// in a binary matrix
import java.io.*;
import java.util.*;

class GFG
{
    // Finds the maximum area under the histogram represented
    // by histogram. See below article for details.
    // https://www.geeksforgeeks.org/largest-rectangle-under-histogram/
    static int maxHist(int R,int C,int row[])
    {
        // Create an empty stack. The stack holds indexes of
        // hist[] array/ The bars stored in stack are always
        // in increasing order of their heights.
        Stack<Integer> result = new Stack<Integer>();

        int top_val;    // Top of stack
    }
```



```
int max_area = 0; // Initialize max area in current
                  // row (or histogram)

int area = 0;    // Initialize area with current top

// Run through all bars of given histogram (or row)
int i = 0;
while (i < C)
{
    // If this bar is higher than the bar on top stack,
    // push it to stack
    if (result.empty() || row[result.peek()] <= row[i])
        result.push(i++);

    else
    {
        // If this bar is lower than top of stack, then
        // calculate area of rectangle with stack top as
        // the smallest (or minimum height) bar. 'i' is
        // 'right index' for the top and element before
        // top in stack is 'left index'
        top_val = row[result.peek()];
        result.pop();
        area = top_val * i;

        if (!result.empty())
            area = top_val * (i - result.peek() - 1 );
        max_area = Math.max(area, max_area);
    }
}

// Now pop the remaining bars from stack and calculate
// area with every popped bar as the smallest bar
while (!result.empty())
{
    top_val = row[result.peek()];
    result.pop();
    area = top_val * i;
    if (!result.empty())
        area = top_val * (i - result.peek() - 1 );

    max_area = Math.max(area, max_area);
}
return max_area;
}

// Returns area of the largest rectangle with all 1s in
// A[] []
```

```
static int maxRectangle(int R,int C,int A[][])
{
    // Calculate area for first row and initialize it as
    // result
    int result = maxHist(R,C,A[0]);

    // iterate over row to find maximum rectangular area
    // considering each row as histogram
    for (int i = 1; i < R; i++)
    {
        for (int j = 0; j < C; j++)

            // if A[i][j] is 1 then add A[i -1][j]
            if (A[i][j] == 1) A[i][j] += A[i - 1][j];

        // Update result if area with current row (as last
        // row of rectangle) is more
        result = Math.max(result, maxHist(R,C,A[i]));
    }

    return result;
}

// Driver code
public static void main (String[] args)
{
    int R = 4;
    int C = 4;

    int A[][] = { {0, 1, 1, 0},
                   {1, 1, 1, 1},
                   {1, 1, 1, 1},
                   {1, 1, 0, 0},
                 };
    System.out.print("Area of maximum rectangle is " +
                     maxRectangle(R,C,A));
}

// Contributed by Prakriti Gupta
```

Output :

Area of maximum rectangle is 8

Time Complexity : $O(R \times X)$

Source

<https://www.geeksforgeeks.org/maximum-size-rectangle-binary-sub-matrix-1s/>

Chapter 70

Maximum sum of smallest and second smallest in an array

Maximum sum of smallest and second smallest in an array - GeeksforGeeks

Given an array, find maximum sum of smallest and second smallest elements chosen from all possible subarrays. More formally, if we write all $(nC2)$ subarrays of array of size ≥ 2 and find the sum of smallest and second smallest, then our answer will be maximum sum among them.

Examples:

```
Input : arr[] = [4, 3, 1, 5, 6]
Output : 11
Subarrays with smallest and second smallest are,
[4, 3]      smallest = 3    second smallest = 4
[4, 3, 1]   smallest = 1    second smallest = 3
[4, 3, 1, 5] smallest = 1    second smallest = 3
[4, 3, 1, 5, 6] smallest = 1    second smallest = 3
[3, 1]      smallest = 1    second smallest = 3
[3, 1, 5]   smallest = 1    second smallest = 3
[3, 1, 5, 6] smallest = 1    second smallest = 3
[1, 5]      smallest = 1    second smallest = 5
[1, 5, 6]   smallest = 1    second smallest = 5
[5, 6]      smallest = 5    second smallest = 6
Maximum sum among all above choices is, 5 + 6 = 11
```

```
Input : arr[] = {5, 4, 3, 1, 6}
Output : 9
```

A **simple solution** is to generate all subarrays, find sum of smallest and second smallest of every subarray. Finally return maximum of all sums.

An **efficient solution** is based on the observation that this problem reduces to finding a maximum sum of two consecutive elements in array.

C++

```
// C++ program to get max sum with smallest
// and second smallest element from any subarray
#include <bits/stdc++.h>
using namespace std;

/* Method returns maximum obtainable sum value
   of smallest and the second smallest value
   taken over all possible subarrays */
int pairWithMaxSum(int arr[], int N)
{
    if (N < 2)
        return -1;

    // Find two consecutive elements with maximum
    // sum.
    int res = arr[0] + arr[1];
    for (int i=1; i<N-1; i++)
        res = max(res, arr[i] + arr[i+1]);

    return res;
}

// Driver code to test above methods
int main()
{
    int arr[] = {4, 3, 1, 5, 6};
    int N = sizeof(arr) / sizeof(int);

    cout << pairWithMaxSum(arr, N) << endl;
    return 0;
}
```

JAVA

```
// Java program to get max sum with smallest
// and second smallest element from any subarray
import java.lang.*;
class num{

    // Method returns maximum obtainable sum value
    // of smallest and the second smallest value
    // taken over all possible subarrays */
```

```
static int pairWithMaxSum(int[] arr, int N)
{
    if (N < 2)
        return -1;

    // Find two consecutive elements with maximum
    // sum.
    int res = arr[0] + arr[1];
    for (int i=1; i<N-1; i++)
        res = Math.max(res, arr[i] + arr[i+1]);

    return res;
}

// Driver program
public static void main(String[] args)
{
    int arr[] = {4, 3, 1, 5, 6};
    int N = arr.length;
    System.out.println(pairWithMaxSum(arr, N));
}
//This code is contributed by
//Smitha Dinesh Semwal
```

Python3

```
# Python 3 program to get max
# sum with smallest and second
# smallest element from any
# subarray

# Method returns maximum obtainable
# sum value of smallest and the
# second smallest value taken
# over all possible subarrays
def pairWithMaxSum(arr, N):

    if (N < 2):
        return -1

    # Find two consecutive elements with
    # maximum sum.
    res = arr[0] + arr[1]

    for i in range(1, N-1):
        res = max(res, arr[i] + arr[i + 1])
```

```
        return res

# Driver code
arr = [4, 3, 1, 5, 6]
N = len(arr)

print(pairWithMaxSum(arr, N))

# This code is contributed by Smitha Dinesh Semwal
```

C#

```
// C# program to get max sum with smallest
// and second smallest element from any subarray
using System;

class GFG {

    // Method returns maximum obtainable sum value
    // of smallest and the second smallest value
    // taken over all possible subarrays
    static int pairWithMaxSum(int []arr, int N)
    {

        if (N < 2)
            return -1;

        // Find two consecutive elements
        // with maximum sum.
        int res = arr[0] + arr[1];
        for (int i = 1; i < N - 1; i++)
            res = Math.Max(res, arr[i] + arr[i + 1]);

        return res;
    }

    // Driver code
    public static void Main()
    {
        int []arr = {4, 3, 1, 5, 6};
        int N = arr.Length;
        Console.Write(pairWithMaxSum(arr, N));
    }
}

// This code is contributed by Nitin Mittal.
```

PHP

```
<?php
// PHP program to get max sum with smallest
// and second smallest element from any subarray

/* Method returns maximum
   obtainable sum value
   of smallest and the
   second smallest value
   taken over all possible
   subarrays */
function pairWithMaxSum( $arr, $N)
{
    if ($N < 2)
        return -1;

    // Find two consecutive
    // elements with maximum
    // sum.
    $res = $arr[0] + $arr[1];
    for($i = 1; $i < $N - 1; $i++)
        $res = max($res, $arr[$i] +
                    $arr[$i + 1]);

    return $res;
}

// Driver Code
$arr = array(4, 3, 1, 5, 6);
$N = count($arr);

echo pairWithMaxSum($arr, $N);

// This code is contributed by anuj_67.
?>
```

Output:

11

Time Complexity : $O(n)$

Thanks to Md Mishfaq Ahmed for suggesting this approach.

Improved By : [nitin mittal](#), [vt_m](#)

Source

<https://www.geeksforgeeks.org/maximum-sum-of-smallest-and-second-smallest-in-an-array/>

Chapter 71

Merge Overlapping Intervals

Merge Overlapping Intervals - GeeksforGeeks

Given a set of time intervals in any order, merge all overlapping intervals into one and output the result which should have only mutually exclusive intervals. Let the intervals be represented as pairs of integers for simplicity.

For example, let the given set of intervals be $\{\{1,3\}, \{2,4\}, \{5,7\}, \{6,8\}\}$. The intervals $\{1,3\}$ and $\{2,4\}$ overlap with each other, so they should be merged and become $\{1, 4\}$. Similarly $\{5, 7\}$ and $\{6, 8\}$ should be merged and become $\{5, 8\}$

Write a function which produces the set of merged intervals for the given set of intervals.

A **simple approach** is to start from the first interval and compare it with all other intervals for overlapping, if it overlaps with any other interval, then remove the other interval from list and merge the other into the first interval. Repeat the same steps for remaining intervals after first. This approach cannot be implemented in better than $O(n^2)$ time.

An **efficient approach** is to first sort the intervals according to starting time. Once we have the sorted intervals, we can combine all intervals in a linear traversal. The idea is, in sorted array of intervals, if $\text{interval}[i]$ doesn't overlap with $\text{interval}[i-1]$, then $\text{interval}[i+1]$ cannot overlap with $\text{interval}[i-1]$ because starting time of $\text{interval}[i+1]$ must be greater than or equal to $\text{interval}[i]$. Following is the detailed step by step algorithm.

1. Sort the intervals based on increasing order of starting time.
2. Push the first interval on to a stack.
3. For each interval do the following
 - a. If the current interval does not overlap with the stack top, push it.
 - b. If the current interval overlaps with stack top and ending time of current interval is more than that of stack top, update stack top with the ending time of current interval.
4. At the end stack contains the merged intervals.

Below is a C++ implementation of the above approach.

```
// A C++ program for merging overlapping intervals
#include<bits/stdc++.h>
using namespace std;

// An interval has start time and end time
struct Interval
{
    int start, end;
};

// Compares two intervals according to their starting time.
// This is needed for sorting the intervals using library
// function std::sort(). See http://goo.gl/iGspV
bool compareInterval(Interval i1, Interval i2)
{
    return (i1.start < i2.start);
}

// The main function that takes a set of intervals, merges
// overlapping intervals and prints the result
void mergeIntervals(Interval arr[], int n)
{
    // Test if the given set has at least one interval
    if (n <= 0)
        return;

    // Create an empty stack of intervals
    stack<Interval> s;

    // sort the intervals in increasing order of start time
    sort(arr, arr+n, compareInterval);

    // push the first interval to stack
    s.push(arr[0]);

    // Start from the next interval and merge if necessary
    for (int i = 1 ; i < n; i++)
    {
        // get interval from stack top
        Interval top = s.top();

        // if current interval is not overlapping with stack top,
        // push it to the stack
        if (top.end < arr[i].start)
            s.push(arr[i]);
    }
}
```

```
// Otherwise update the ending time of top if ending of current
// interval is more
else if (top.end < arr[i].end)
{
    top.end = arr[i].end;
    s.pop();
    s.push(top);
}

// Print contents of stack
cout << "\n The Merged Intervals are: ";
while (!s.empty())
{
    Interval t = s.top();
    cout << "[" << t.start << "," << t.end << "]" ";
    s.pop();
}
return;
}

// Driver program
int main()
{
    Interval arr[] = { {6,8}, {1,9}, {2,4}, {4,7} };
    int n = sizeof(arr)/sizeof(arr[0]);
    mergeIntervals(arr, n);
    return 0;
}
```

Output:

The Merged Intervals are: [1,9]

Time complexity of the method is $O(n\log n)$ which is for sorting. Once the array of intervals is sorted, merging takes linear time.

A $O(n \log n)$ and $O(1)$ Extra Space Solution

The above solution requires $O(n)$ extra space for stack. We can avoid use of extra space by doing merge operations in-place. Below are detailed steps.

- 1) Sort all intervals in decreasing order of start time.
- 2) Traverse sorted intervals starting from first interval, do following for every interval.
 - a) If current interval is not first interval and it overlaps with previous interval, then merge it with previous interval. Keep doing it while the interval

- overlaps with the previous one.
- b) Else add current interval to output list of intervals.

Note that if intervals are sorted by decreasing order of start times, we can quickly check if intervals overlap or not by comparing start time of previous interval with end time of current interval.

Below is C++ implementation of above algorithm.

```
// C++ program to merge overlapping Intervals in
// O(n Log n) time and O(1) extra space.
#include<bits/stdc++.h>
using namespace std;

// An Interval
struct Interval
{
    int s, e;
};

// Function used in sort
bool mycomp(Interval a, Interval b)
{    return a.s > b.s; }

void mergeIntervals(Interval arr[], int n)
{
    // Sort Intervals in decreasing order of
    // start time
    sort(arr, arr+n, mycomp);

    int index = 0; // Stores index of last element
    // in output array (modified arr[])

    // Traverse all input Intervals
    for (int i=0; i<n; i++)
    {
        // If this is not first Interval and overlaps
        // with the previous one
        if (index != 0 && arr[index-1].s <= arr[i].e)
        {
            while (index != 0 && arr[index-1].s <= arr[i].e)
            {
                // Merge previous and current Intervals
                arr[index-1].e = max(arr[index-1].e, arr[i].e);
                arr[index-1].s = min(arr[index-1].s, arr[i].s);
                index--;
            }
        }
    }
}
```

```
        else // Doesn't overlap with previous, add to
            // solution
            arr[index] = arr[i];

        index++;
    }

    // Now arr[0..index-1] stores the merged Intervals
    cout << "\n The Merged Intervals are: ";
    for (int i = 0; i < index; i++)
        cout << "[" << arr[i].s << ", " << arr[i].e << "] ";
}

// Driver program
int main()
{
    Interval arr[] = { {6,8}, {1,9}, {2,4}, {4,7} };
    int n = sizeof(arr)/sizeof(arr[0]);
    mergeIntervals(arr, n);
    return 0;
}
```

Output:

The Merged Intervals are: [1,9]

Thanks to Gaurav Ahirwar for suggesting this method.

This article is compiled by Ravi Chandra Enaganti. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

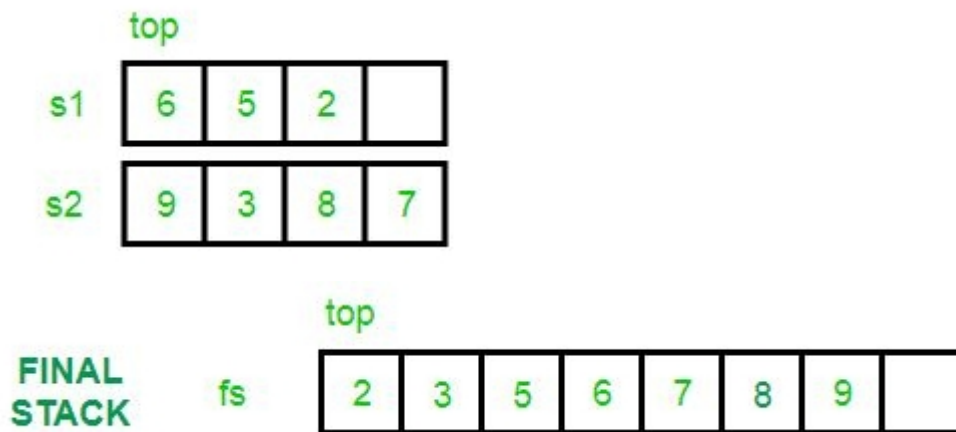
<https://www.geeksforgeeks.org/merging-intervals/>

Chapter 72

Merging and Sorting Two Unsorted Stacks

Merging and Sorting Two Unsorted Stacks - GeeksforGeeks

Given 2 input stacks with elements in an unsorted manner. Problem is to merge them into a new final stack, such that the elements become arranged in a sorted manner.



Examples:

```
Input : s1 : 9 4 2 1
        s2: 8 17 3 10
Output : final stack: 1 2 3 4 8 9 10 17
```

```
Input : s1 : 5 7 2 6 4
        s2 : 12 9 3
```

Output : final stack: 2 3 4 5 6 7 9 12

Create an empty stack to store result. We first insert elements of both stacks into the result. Then we [sort the result stack](#).

C++

```
// C++ program to merge to unsorted stacks
// into a third stack in sorted way.
#include <bits/stdc++.h>
using namespace std;

// Sorts input stack and returns sorted stack.
stack<int> sortStack(stack<int>& input)
{
    stack<int> tmpStack;

    while (!input.empty()) {
        // pop out the first element
        int tmp = input.top();
        input.pop();

        // while temporary stack is not empty and top
        // of stack is greater than temp
        while (!tmpStack.empty() && tmpStack.top() > tmp) {

            // pop from temporary stack and push
            // it to the input stack
            input.push(tmpStack.top());
            tmpStack.pop();
        }

        // push temp in tempory of stack
        tmpStack.push(tmp);
    }

    return tmpStack;
}

stack<int> sortedMerge(stack<int>& s1, stack<int>& s2)
{
    // Push contents of both stacks in result
    stack<int> res;
    while (!s1.empty()) {
        res.push(s1.top());
        s1.pop();
    }
    while (!s2.empty()) {
```

```
        res.push(s2.top());
        s2.pop();
    }

    // Sort the result stack.
    return sortStack(res);
}

// main function
int main()
{
    stack<int> s1, s2;
    s1.push(34);
    s1.push(3);
    s1.push(31);

    s2.push(1);
    s2.push(12);
    s2.push(23);

    // This is the temporary stack
    stack<int> tmpStack = sortedMerge(s1, s2);
    cout << "Sorted and merged stack :\n";

    while (!tmpStack.empty()) {
        cout << tmpStack.top() << " ";
        tmpStack.pop();
    }
}
```

Java

```
// Java program to merge two unsorted stacks
// into a third stack in sorted way.
import java.io.*;
import java.util.*;

public class GFG {

    // This is the temporary stack
    static Stack<Integer> res = new Stack<Integer>();
    static Stack<Integer> tmpStack = new Stack<Integer>();

    // Sorts input stack and returns
    // sorted stack.
    static void sortStack(Stack<Integer> input)
    {
        while (input.size() != 0)
```



```
{
    // pop out the first element
    int tmp = input.peek();
    input.pop();

    // while temporary stack is not empty and
    // top of stack is greater than temp
    while (tmpStack.size() != 0 &&
           tmpStack.peek() > tmp)
    {

        // pop from temporary stack and push
        // it to the input stack
        input.push(tmpStack.peek());
        tmpStack.pop();
    }

    // push temp in temporary of stack
    tmpStack.push(tmp);
}

static void sortedMerge(Stack<Integer> s1,
                        Stack<Integer> s2)
{
    // Push contents of both stacks in result
    while (s1.size() != 0) {
        res.push(s1.peek());
        s1.pop();
    }

    while (s2.size() != 0) {
        res.push(s2.peek());
        s2.pop();
    }

    // Sort the result stack.
    sortStack(res);
}

// main function
public static void main(String args[])
{
    Stack<Integer> s1 = new Stack<Integer>();
    Stack<Integer> s2 = new Stack<Integer>();
    s1.push(34);
    s1.push(3);
    s1.push(31);
}
```

```
s2.push(1);
s2.push(12);
s2.push(23);

sortedMerge(s1, s2);
System.out.println("Sorted and merged stack :");

while (tmpStack.size() != 0) {
    System.out.print(tmpStack.peek() + " ");
    tmpStack.pop();
}
}

// This code is contributed by Manish Shaw
// (manishshaw1)
```

C#

```
// C# program to merge to unsorted stacks
// into a third stack in sorted way.
using System;
using System.Collections.Generic;

class GFG {

    // Sorts input stack and returns
    // sorted stack.
    static Stack<int> sortStack(ref Stack<int> input)
    {
        Stack<int> tmpStack = new Stack<int>();

        while (input.Count != 0)
        {
            // pop out the first element
            int tmp = input.Peek();
            input.Pop();

            // while temporary stack is not empty and
            // top of stack is greater than tmp
            while (tmpStack.Count != 0 &&
                    tmpStack.Peek() > tmp)
            {
                // pop from temporary stack and push
                // it to the input stack
                input.Push(tmpStack.Peek());
            }
        }
    }
}
```

```
        tmpStack.Pop();
    }

    // push temp in temporary of stack
    tmpStack.Push(tmp);
}

return tmpStack;
}

static Stack<int> sortedMerge(ref Stack<int> s1,
                              ref Stack<int> s2)
{
    // Push contents of both stacks in result
    Stack<int> res = new Stack<int>();
    while (s1.Count!=0) {
        res.Push(s1.Peek());
        s1.Pop();
    }
    while (s2.Count!=0) {
        res.Push(s2.Peek());
        s2.Pop();
    }

    // Sort the result stack.
    return sortStack(ref res);
}

// main function
static void Main()
{
    Stack<int> s1 = new Stack<int>();
    Stack<int> s2 = new Stack<int>();
    s1.Push(34);
    s1.Push(3);
    s1.Push(31);

    s2.Push(1);
    s2.Push(12);
    s2.Push(23);

    // This is the temporary stack
    Stack<int> tmpStack = new Stack<int>();
    tmpStack = sortedMerge(ref s1,ref s2);
    Console.WriteLine("Sorted and merged stack :\n");

    while (tmpStack.Count!=0) {
        Console.WriteLine(tmpStack.Peek() + " ");
    }
}
```

```
        tmpStack.Pop();
    }
}

// This code is contributed by Manish Shaw
// (manishshaw1)
```

Output:

Sorted and merged stack :
34 31 23 12 3 1

Improved By : [manishshaw1](#)

Source

<https://www.geeksforgeeks.org/merging-sorting-two-unsorted-stacks/>

Chapter 73

Minimum number of bracket reversals needed to make an expression balanced

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

Given an expression with only '}' and '{'. The expression may not be balanced. Find minimum number of bracket reversals to make the expression balanced.

Examples:

Input: exp = "}{"

Output: 2

We need to change '}' to '{' and '{' to '}' so that the expression becomes balanced, the balanced expression is '{}'

Input: exp = "{{{"

Output: Can't be made balanced using reversals

Input: exp = "{{{{{

Output: 2

Input: exp = "{{{{{}}}"

Output: 1

Input: exp = "}}{{}}{{{{{"

Output: 3

One simple observation is, the string can be balanced only if total number of brackets is even (there must be equal no of '{' and '}')

A **Naive Solution** is to consider every bracket and recursively count number of reversals by taking two cases (i) keeping the bracket as it is (ii) reversing the bracket. If we get a balanced expression, we update result if number of steps followed for reaching here is smaller than the minimum so far. Time complexity of this solution is $O(2^n)$.

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.

Below is implementation of above idea.

C++

```
// C++ program to find minimum number of
// reversals required to balance an expression
#include<bits/stdc++.h>
using namespace std;

// Returns count of minimum reversals for making
// expr balanced. Returns -1 if expr cannot be
// balanced.
int countMinReversals(string expr)
{
    int len = expr.length();

    // length of expression must be even to make
    // it balanced by using reversals.
    if (len%2)
        return -1;

    // After this loop, stack contains unbalanced
    // part of expression, i.e., expression of the
    // form "}}...}{...{"
    stack<char> s;
    for (int i=0; i<len; i++)
    {
        if (expr[i]=='}' && !s.empty())
        {
            if (s.top()=='{')
                s.pop();
        }
    }
```

```
        else
            s.push(expr[i]);
    }
    else
        s.push(expr[i]);
}

// Length of the reduced expression
// red_len = (m+n)
int red_len = s.size();

// count opening brackets at the end of
// stack
int n = 0;
while (!s.empty() && s.top() == '{')
{
    s.pop();
    n++;
}

// return ceil(m/2) + ceil(n/2) which is
// actually equal to (m+n)/2 + n%2 when
// m+n is even.
return (red_len/2 + n%2);
}

// Driver program to test above function
int main()
{
    string expr = "}}{{";
    cout << countMinReversals(expr);
    return 0;
}
```

Java

```
//Java Code to count minimum reversal for
//making an expression balanced.

import java.util.Stack;

public class GFG
{
    // Method count minimum reversal for
    //making an expression balanced.
    //Returns -1 if expression cannot be balanced
    static int countMinReversals(String expr)
```

```
{
    int len = expr.length();

    // length of expression must be even to make
    // it balanced by using reversals.
    if (len%2 != 0)
        return -1;

    // After this loop, stack contains unbalanced
    // part of expression, i.e., expression of the
    // form "}}..{{..{"
    Stack<Character> s=new Stack<>();

    for (int i=0; i<len; i++)
    {
        char c = expr.charAt(i);
        if (c =='}' && !s.empty())
        {
            if (s.peek()=='{')
                s.pop();
            else
                s.push(c);
        }
        else
            s.push(c);
    }

    // Length of the reduced expression
    // red_len = (m+n)
    int red_len = s.size();

    // count opening brackets at the end of
    // stack
    int n = 0;
    while (!s.empty() && s.peek() == '{')
    {
        s.pop();
        n++;
    }

    // return ceil(m/2) + ceil(n/2) which is
    // actually equal to (m+n)/2 + n%2 when
    // m+n is even.
    return (red_len/2 + n%2);
}

// Driver method
public static void main(String[] args)
```



```
{
    String expr = "}}{{";

    System.out.println(countMinReversals(expr));
}

//This code is contributed by Sumit Ghosh
```

Output:

2

Time Complexity: $O(n)$

Auxiliary Space: $O(n)$

Thanks to Utkarsh Trivedi for suggesting above approach.

Source

<https://www.geeksforgeeks.org/minimum-number-of-bracket-reversals-needed-to-make-an-expression-balanced/>

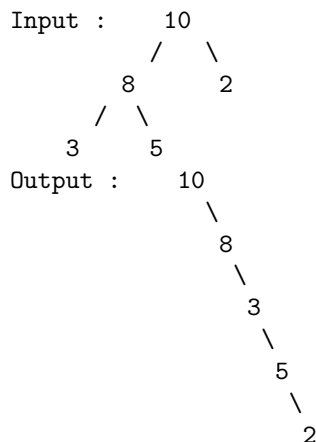
Chapter 74

Modify a binary tree to get preorder traversal using right pointers only

Modify a binary tree to get preorder traversal using right pointers only - GeeksforGeeks

Given a binary tree. Modify it in such a way that after modification you can have a preorder traversal of it using only the right pointers. During modification, you can use right as well as left pointers.

Examples:



Explanation : The preorder traversal of given binary tree is 10 8 3 5 2.

Method 1 (Recursive)

One needs to make the right pointer of root point to the left subtree.

If the node has just left child, then just moving the child to right will complete the processing for that node.

If there is a right child too, then it should be made **right child of the right-most of the original left subtree**.

The above function used in the code process a node and then returns the rightmost node of the transformed subtree.

C++

```
// C code to modify binary tree for
// traversal using only right pointer
#include <iostream>
#include <stack>
#include <stdio.h>
#include <stdlib.h>

using namespace std;

// A binary tree node has data,
// left child and right child
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// function that allocates a new node
// with the given data and NULL left
// and right pointers.
struct Node* newNode(int data)
{
    struct Node* node = new struct Node;
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return (node);
}

// Function to modify tree
struct Node* modifytree(struct Node* root)
{
    struct Node* right = root->right;
    struct Node* rightMost = root;

    // if the left tree exists
    if (root->left) {

        // get the right-most of the
        // original left subtree
    }
}
```

```
        rightMost = modifytree(root->left);

        // set root right to left subtree
        root->right = root->left;
        root->left = NULL;
    }

    // if the right subtree does
    // not exists we are done!
    if (!right)
        return rightMost;

    // set right pointer of right-most
    // of the original left subtree
    rightMost->right = right;

    // modify the rightsubtree
    rightMost = modifytree(right);
    return rightMost;
}

// printing using right pointer only
void printpre(struct Node* root)
{
    while (root != NULL) {
        cout << root->data << " ";
        root = root->right;
    }
}

// Driver program to test above functions
int main()
{
    /* Constructed binary tree is
        10
       / \
      8   2
     / \
    3   5   */
    struct Node* root = newNode(10);
    root->left = newNode(8);
    root->right = newNode(2);
    root->left->left = newNode(3);
    root->left->right = newNode(5);

    modifytree(root);
    printpre(root);
}
```

```
    return 0;
}
```

Output:

```
10 8 3 5 2
```

Method 2 (Iterative)

This can be easily done using iterative preorder traversal. See here. [Iterative preorder traversal](#)

The idea is to maintain a variable prev which maintains the previous node of the preorder traversal. Every-time a new node is encountered, the node set its right to previous one and prev is made equal to the current node. In the end we will have a sort of linked list whose first element is root then left child then right, so on and so forth.

C++

```
// C code to modify binary tree for
// traversal using only right pointer
#include <iostream>
#include <stack>
#include <stdio.h>
#include <stdlib.h>

using namespace std;

// A binary tree node has data,
// left child and right child
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Helper function that allocates a new
// node with the given data and NULL
// left and right pointers.
struct Node* newNode(int data)
{
    struct Node* node = new struct Node;
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return (node);
}

// An iterative process to set the right
```

```
// pointer of Binary tree
void modifytree(struct Node* root)
{
    // Base Case
    if (root == NULL)
        return;

    // Create an empty stack and push root to it
    stack<Node*> nodeStack;
    nodeStack.push(root);

    /* Pop all items one by one.
       Do following for every popped item
       a) print it
       b) push its right child
       c) push its left child
       Note that right child is pushed first
       so that left is processed first */
    struct Node* pre = NULL;
    while (nodeStack.empty() == false) {

        // Pop the top item from stack
        struct Node* node = nodeStack.top();

        nodeStack.pop();

        // Push right and left children of
        // the popped node to stack
        if (node->right)
            nodeStack.push(node->right);
        if (node->left)
            nodeStack.push(node->left);

        // check if some previous node exists
        if (pre != NULL) {

            // set the right pointer of
            // previous node to current
            pre->right = node;
        }

        // set previous node as current node
        pre = node;
    }
}

// printing using right pointer only
void printpre(struct Node* root)
```

```
{
    while (root != NULL) {
        cout << root->data << " ";
        root = root->right;
    }
}

// Driver code
int main()
{
    /* Constructed binary tree is
          10
         /  \
        8    2
       /  \
      3    5
    */
    struct Node* root = newNode(10);
    root->left = newNode(8);
    root->right = newNode(2);
    root->left->left = newNode(3);
    root->left->right = newNode(5);

    modifytree(root);
    printpre(root);

    return 0;
}
```

Output:

10 8 3 5 2

Improved By : [02DCE](#)

Source

<https://www.geeksforgeeks.org/modify-binary-tree-get-preorder-traversal-using-right-pointers/>

Chapter 75

Next Greater Element

Next Greater Element - GeeksforGeeks

Given an array, print the Next Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in array. Elements for which no greater element exist, consider next greater element as -1.

Examples:

- a) For any array, rightmost element always has next greater element as -1.
- b) For an array which is sorted in decreasing order, all elements have next greater element as -1.
- c) For the input array [4, 5, 2, 25], the next greater elements for each element are as follows.

Element		NGE
4	-->	5
5	-->	25
2	-->	25
25	-->	-1

- d) For the input array [13, 7, 6, 12], the next greater elements for each element are as follows.

Element		NGE
13	-->	-1
7	-->	12
6	-->	12
12	-->	-1

Method 1 (Simple)

Use two loops: The outer loop picks all the elements one by one. The inner loop looks for

the first greater element for the element picked by outer loop. If a greater element is found then that element is printed as next, otherwise -1 is printed.

Thanks to Sachin for providing following code.

C

```
// Simple C program to print next greater elements
// in a given array
#include<stdio.h>

/* prints element and NGE pair for all elements of
arr[] of size n */
void printNGE(int arr[], int n)
{
    int next, i, j;
    for (i=0; i<n; i++)
    {
        next = -1;
        for (j = i+1; j<n; j++)
        {
            if (arr[i] < arr[j])
            {
                next = arr[j];
                break;
            }
        }
        printf("%d -- %dn", arr[i], next);
    }
}

int main()
{
    int arr[] = {11, 13, 21, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    printNGE(arr, n);
    return 0;
}
```

Java

```
// Simple Java program to print next
// greater elements in a given array

class Main
{
    /* prints element and NGE pair for
```

```
    all elements of arr[] of size n */
static void printNGE(int arr[], int n)
{
    int next, i, j;
    for (i=0; i<n; i++)
    {
        next = -1;
        for (j = i+1; j<n; j++)
        {
            if (arr[i] < arr[j])
            {
                next = arr[j];
                break;
            }
        }
        System.out.println(arr[i]+" -- "+next);
    }
}

public static void main(String args[])
{
    int arr[] = {11, 13, 21, 3};
    int n = arr.length;
    printNGE(arr, n);
}
}
```

Python

```
# Function to print element and NGE pair for all elements of list
def printNGE(arr):

    for i in range(0, len(arr), 1):

        next = -1
        for j in range(i+1, len(arr), 1):
            if arr[i] < arr[j]:
                next = arr[j]
                break

        print(str(arr[i]) + " -- " + str(next))

# Driver program to test above function
arr = [11,13,21,3]
printNGE(arr)

# This code is contributed by Sunny Karira
```

C#

```
// Simple C# program to print next
// greater elements in a given array
using System;

class GFG
{
    /* prints element and NGE pair for
    all elements of arr[] of size n */
    static void printNGE(int []arr, int n)
    {
        int next, i, j;
        for (i = 0; i < n; i++)
        {
            next = -1;
            for (j = i + 1; j < n; j++)
            {
                if (arr[i] < arr[j])
                {
                    next = arr[j];
                    break;
                }
            }
            Console.WriteLine(arr[i] + " -- " + next);
        }
    }

    // driver code
    public static void Main()
    {
        int []arr= {11, 13, 21, 3};
        int n = arr.Length;

        printNGE(arr, n);
    }
}

// This code is contributed by Sam007
```

PHP

```
<?php
// Simple PHP program to print next
// greater elements in a given array
```

```
/* prints element and NGE pair for
   all elements of arr[] of size n */
function printNGE($arr, $n)
{
    for ($i = 0; $i < $n; $i++)
    {
        $next = -1;
        for ($j = $i + 1; $j < $n; $j++)
        {
            if ($arr[$i] < $arr[$j])
            {
                $next = $arr[$j];
                break;
            }
        }
        echo $arr[$i]. " -- " . $next. "\n";
    }
}

// Driver Code
$arr= array(11, 13, 21, 3);
$n = count($arr);
printNGE($arr, $n);

// This code is contributed by Sam007
?>
```

Output:

```
11 -- 13
13 -- 21
21 -- -1
3 -- -1
```

Time Complexity: $O(n^2)$. The worst case occurs when all elements are sorted in decreasing order.

Method 2 (Using Stack)

- 1) Push the first element to stack.
- 2) Pick rest of the elements one by one and follow following steps in loop.
 -a) Mark the current element as *next*.
 -b) If stack is not empty, then pop an element from stack and compare it with *next*.
 -c) If *next* is greater than the popped element, then *next* is the next greater element for the popped element.
 -d) Keep popping from the stack while the popped element is smaller than *next*. *next* becomes the next greater element for all such popped elements

3) After the loop in step 2 is over, pop all the elements from stack and print -1 as next element for them.

C++

```
// A Stack based C++ program to find next
// greater element for all array elements.
#include <bits/stdc++.h>
using namespace std;

/* prints element and NGE pair for all
elements of arr[] of size n */
void printNGE(int arr[], int n) {
    stack < int > s;

    /* push the first element to stack */
    s.push(arr[0]);

    /* iterate for rest of the elements
    for (int i = 1; i < n; i++) {

        if (s.empty()) {
            s.push(arr[i]);
            continue;
        }

        /* if stack is not empty, then
        pop an element from stack.
        If the popped element is smaller
        than next, then
        a) print the pair
        b) keep popping while elements are
        smaller and stack is not empty */
        while (s.empty() == false && s.top() < arr[i])
        {
            cout << s.top() << " --> " << arr[i] << endl;
            s.pop();
        }

        /* push next to stack so that we can find
        next greater for it */
        s.push(arr[i]);
    }

    /* After iterating over the loop, the remaining
    elements in stack do not have the next greater
    element, so print -1 for them */
    while (s.empty() == false) {
```

```
        cout << s.top() << " --> " << -1 << endl;
        s.pop();
    }
}

/* Driver program to test above functions */
int main() {
    int arr[] = {11, 13, 21, 3};
    int n = sizeof(arr) / sizeof(arr[0]);
    printNGE(arr, n);
    return 0;
}
```

C

```
// A Stack based C program to find next greater element
// for all array elements.
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#define STACKSIZE 100

// stack structure
struct stack
{
    int top;
    int items[STACKSIZE];
};

// Stack Functions to be used by printNGE()
void push(struct stack *ps, int x)
{
    if (ps->top == STACKSIZE-1)
    {
        printf("Error: stack overflown");
        getchar();
        exit(0);
    }
    else
    {
        ps->top += 1;
        int top = ps->top;
        ps->items [top] = x;
    }
}

bool isEmpty(struct stack *ps)
{

```

```
    return (ps->top == -1)? true : false;
}

int pop(struct stack *ps)
{
    int temp;
    if (ps->top == -1)
    {
        printf("Error: stack underflow n");
        getchar();
        exit(0);
    }
    else
    {
        int top = ps->top;
        temp = ps->items [top];
        ps->top -= 1;
        return temp;
    }
}

/* prints element and NGE pair for all elements of
arr[] of size n */
void printNGE(int arr[], int n)
{
    int i = 0;
    struct stack s;
    s.top = -1;
    int element, next;

    /* push the first element to stack */
    push(&s, arr[0]);

    // iterate for rest of the elements
    for (i=1; i<n; i++)
    {
        next = arr[i];

        if (isEmpty(&s) == false)
        {
            // if stack is not empty, then pop an element from stack
            element = pop(&s);

            /* If the popped element is smaller than next, then
            a) print the pair
            b) keep popping while elements are smaller and
            stack is not empty */
            while (element < next)
```

```
        {
            printf("n %d --> %d", element, next);
            if(isEmpty(&s) == true)
                break;
            element = pop(&s);
        }

        /* If element is greater than next, then push
        the element back */
        if (element > next)
            push(&s, element);
    }

    /* push next to stack so that we can find
    next greater for it */
    push(&s, next);
}

/* After iterating over the loop, the remaining
elements in stack do not have the next greater
element, so print -1 for them */
while (isEmpty(&s) == false)
{
    element = pop(&s);
    next = -1;
    printf("n %d --> %d", element, next);
}
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {11, 13, 21, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    printNGE(arr, n);
    getchar();
    return 0;
}
```

Java

```
//Java program to print next
//greater element using stack

public class NGE
{
    static class stack
    {
```



```
int top;
int items[] = new int[100];

// Stack functions to be used by printNGE
void push(int x)
{
    if (top == 99)
    {
        System.out.println("Stack full");
    }
    else
    {
        items[++top] = x;
    }
}

int pop()
{
    if (top == -1)
    {
        System.out.println("Underflow error");
        return -1;
    }
    else
    {
        int element = items[top];
        top--;
        return element;
    }
}

boolean isEmpty()
{
    return (top == -1) ? true : false;
}

/* prints element and NGE pair for
all elements of arr[] of size n */
static void printNGE(int arr[], int n)
{
    int i = 0;
    stack s = new stack();
    s.top = -1;
    int element, next;

    /* push the first element to stack */
    s.push(arr[0]);
```

```
// iterate for rest of the elements
for (i = 1; i < n; i++)
{
    next = arr[i];

    if (s.isEmpty() == false)
    {

        // if stack is not empty, then
        // pop an element from stack
        element = s.pop();

        /* If the popped element is smaller than
        next, then a) print the pair b) keep
        popping while elements are smaller and
        stack is not empty */
        while (element < next)
        {
            System.out.println(element + " --> " + next);
            if (s.isEmpty() == true)
                break;
            element = s.pop();
        }

        /* If element is greater than next, then
        push the element back */
        if (element > next)
            s.push(element);
    }

    /* push next to stack so that we can find next
    greater for it */
    s.push(next);
}

/* After iterating over the loop, the remaining
elements in stack do not have the next greater
element, so print -1 for them */
while (s.isEmpty() == false)
{
    element = s.pop();
    next = -1;
    System.out.println(element + " -- " + next);
}

}

public static void main(String[] args)
```

```
{
    int arr[] = { 11, 13, 21, 3 };
    int n = arr.length;
    printNGE(arr, n);
}

// Thanks to Rishabh Mahrsee for contributing this code
```

Python

```
# Python program to print next greater element using stack

# Stack Functions to be used by printNGE()
def createStack():
    stack = []
    return stack

def isEmpty(stack):
    return len(stack) == 0

def push(stack, x):
    stack.append(x)

def pop(stack):
    if isEmpty(stack):
        print("Error : stack underflow")
    else:
        return stack.pop()

'''prints element and NGE pair for all elements of
arr[] '''
def printNGE(arr):
    s = createStack()
    element = 0
    next = 0

    # push the first element to stack
    push(s, arr[0])

    # iterate for rest of the elements
    for i in range(1, len(arr), 1):
        next = arr[i]

        if isEmpty(s) == False:

            # if stack is not empty, then pop an element from stack
            element = pop(s)
```

```
        '''If the popped element is smaller than next, then
        a) print the pair
        b) keep popping while elements are smaller and
           stack is not empty '''
    while element < next :
        print(str(element)+ " -- " + str(next))
        if isEmpty(s) == True :
            break
        element = pop(s)

    '''If element is greater than next, then push
    the element back '''
    if element > next:
        push(s, element)

    '''push next to stack so that we can find
    next greater for it '''
    push(s, next)

    '''After iterating over the loop, the remaining
    elements in stack do not have the next greater
    element, so print -1 for them '''

    while isEmpty(s) == False:
        element = pop(s)
        next = -1
        print(str(element) + " -- " + str(next))

# Driver program to test above functions
arr = [11, 13, 21, 3]
printNGE(arr)

# This code is contributed by Sunny Karira
```

Output:

```
11 -- 13
13 -- 21
3 -- -1
21 -- -1
```

Time Complexity: $O(n)$. The worst case occurs when all elements are sorted in decreasing order. If elements are sorted in decreasing order, then every element is processed at most 4 times.

a) Initially pushed to the stack.

- b) Popped from the stack when next element is being processed.
- c) Pushed back to the stack because next element is smaller.
- d) Popped from the stack in step 3 of algo.

How to get elements in same order as input?

The above approach may not produce output elements in same order as input. To achieve same order, we can traverse the same in reverse order

C++

```
// A Stack based C++ program to find next
// greater element for all array elements
// in same order as input.
#include <bits/stdc++.h>

using namespace std;

/* prints element and NGE pair for all
elements of arr[] of size n */
void printNGE(int arr[], int n)
{
    stack<int> s;
    unordered_map<int, int> mp;

    /* push the first element to stack */
    s.push(arr[0]);

    // iterate for rest of the elements
    for (int i = 1; i < n; i++) {

        if (s.empty()) {
            s.push(arr[i]);
            continue;
        }

        /* if stack is not empty, then
        pop an element from stack.
        If the popped element is smaller
        than next, then
        a) print the pair
        b) keep popping while elements are
        smaller and stack is not empty */
        while (s.empty() == false && s.top() < arr[i]) {
            mp[s.top()] = arr[i];
            s.pop();
        }
    }
}
```

```
        /* push next to stack so that we can find
        next smaller for it */
        s.push(arr[i]);
    }

    /* After iterating over the loop, the remaining
    elements in stack do not have the next smaller
    element, so print -1 for them */
    while (s.empty() == false) {
        mp[s.top()] = -1;
        s.pop();
    }

    for (int i=0; i<n; i++)
        cout << arr[i] << " ---> " << mp[arr[i]] << endl;
}

/* Driver program to test above functions */
int main()
{
    int arr[] = { 11, 13, 21, 3 };
    int n = sizeof(arr) / sizeof(arr[0]);
    printNGE(arr, n);
    return 0;
}
```

Java

```
// A Stack based Java program to find next
// greater element for all array elements
// in same order as input.
import java.util.Stack;

class NextGreaterElement
{
    private int arr[] = {11, 13, 21, 3};

    /* prints element and NGE pair for all
    elements of arr[] of size n */
    private void printNGE()
    {
        Stack<Integer> s = new Stack<>();
        int nge[] = new int[arr.length];

        // iterate for rest of the elements
        for (int i = arr.length - 1; i >= 0; i--) {
```

```
        /* if stack is not empty, then
        pop an element from stack.
        If the popped element is smaller
        than next, then
        a) print the pair
        b) keep popping while elements are
        smaller and stack is not empty */
        if (!s.empty()) {
            while (!s.empty() && s.peek() <= arr[i]) {
                s.pop();
            }
        }
        nge[i] = s.empty() ? -1 : s.peek();
        s.push(arr[i]);

    }
    for(int i = 0; i < arr.length; i++)
        System.out.print(nge[i] + " ");
}

/* Driver program to test above functions */
public static void main(String[] args)
{
    NextGreaterElement nge = new NextGreaterElement();
    nge.printNGE();
}

// This code is contributed by Ashish Goyal
```

Improved By : [Sam007](#), [ashishfk](#)

Source

<https://www.geeksforgeeks.org/next-greater-element/>

Chapter 76

Next Greater Frequency Element

Next Greater Frequency Element - GeeksforGeeks

Given an array, for each element find the value of nearest element to the right which is having frequency greater than as that of current element. If there does not exist an answer for a position, then make the value '-1'.

Examples:

Input : a[] = [1, 1, 2, 3, 4, 2, 1]

Output : [-1, -1, 1, 2, 2, 1, -1]

Explanation:

Given array a[] = [1, 1, 2, 3, 4, 2, 1]

Frequency of each element is: 3, 3, 2, 1, 1, 2, 3

Lets calls Next Greater Frequency element as NGF

1. For element a[0] = 1 which has a frequency = 3,
As it has frequency of 3 and no other next element has frequency more than 3 so '-1'
2. For element a[1] = 1 it will be -1 same logic like a[0]
3. For element a[2] = 2 which has frequency = 2,
NGF element is 1 at position = 6 with frequency of 3 > 2
4. For element a[3] = 3 which has frequency = 1,
NGF element is 2 at position = 5 with frequency of 2 > 1
5. For element a[4] = 4 which has frequency = 1,
NGF element is 2 at position = 5 with frequency of 2 > 1
6. For element a[5] = 2 which has frequency = 2,

NGF element is 1 at position = 6 with frequency
of 3 > 2

7. For element a[6] = 1 there is no element to its
right, hence -1

Input : a[] = [1, 1, 1, 2, 2, 2, 2, 11, 3, 3]

Output : [2, 2, 2, -1, -1, -1, -1, 3, -1, -1]

Naive approach:

A simple hashing technique is to use values as index is be used to store frequency of each element. Create a list suppose to store frequency of each number in the array. (Single traversal is required). Now use two loops.

The outer loop picks all the elements one by one.

The inner loop looks for the first element whose frequency is greater than the frequency of current element.

If a greater frequency element is found then that element is printed, otherwise -1 is printed.

Time complexity : $O(n*n)$

Efficient approach:

We can use hashing and stack data structure to efficiently solve for many cases. A simple hashing technique is to use values as index and frequency of each element as value. We use stack data structure to store position of elements in the array.

- 1) Create a list to to use values as index to store frequency of each element.
- 2) Push the position of first element to stack.
- 3) Pick rest of the position of elements one by one and follow following steps in loop.
 -a) Mark the position of current element as 'i' .
 - b) If the frequency of the element which is pointed by the top of stack is **greater** than frequency of the current element, push the current position i to the stack
 - c) If the frequency of the element which is pointed by the top of stack is **less** than frequency of the current element and the stack is not empty then follow these steps:
 -i) continue popping the stack
 -ii) if the condition in step c fails then push the current position i to the stack
- 4) After the loop in step 3 is over, pop all the elements from stack and print -1 as next greater frequency element for them does not exist.

Time complexity is $O(n)$.

Below is the Python 3 implementation of the above problem.

```
'''NFG function to find the next greater frequency
element for each element in the array'''
def NFG(a, n):

    if (n <= 0):
```

```
    print("List empty")
    return []

# stack data structure to store the position
# of array element
stack = [0]*n

# freq is a dictionary which maintains the
# frequency of each element
freq = {}
for i in a:
    freq[a[i]] = 0
for i in a:
    freq[a[i]] += 1

# res to store the value of next greater
# frequency element for each element
res = [0]*n

# initialize top of stack to -1
top = -1

# push the first position of array in the stack
top += 1
stack[top] = 0

# now iterate for the rest of elements
for i in range(1, n):

    ''' If the frequency of the element which is
        pointed by the top of stack is greater
        than frequency of the current element
        then push the current position i in stack'''
    if (freq[a[stack[top]]] > freq[a[i]]):
        top += 1
        stack[top] = i

    else:

        ''' If the frequency of the element which
            is pointed by the top of stack is less
            than frequency of the current element, then
            pop the stack and continuing popping until
            the above condition is true while the stack
            is not empty'''

        while (top>-1 and freq[a[stack[top]]] < freq[a[i]]):
            res[stack[top]] = a[i]
            top -= 1
```

```
        # now push the current element
        top+=1
        stack[top] = i

    '''After iterating over the loop, the remaining
    position of elements in stack do not have the
    next greater element, so print -1 for them'''
    while (top > -1):
        res[stack[top]] = -1
        top -= 1

    # return the res list containing next
    # greater frequency element
    return res

# Driver program to test the function
print(NFG([1,1,2,3,4,2,1],7))
```

Output:

```
[-1, -1, 1, 2, 2, 1, -1]
```

Source

<https://www.geeksforgeeks.org/next-greater-frequency-element/>

Chapter 77

Next Smaller Element

Next Smaller Element - GeeksforGeeks

Given an array, print the Next Smaller Element (NSE) for every element. The Smaller smaller Element for an element x is the first smaller element on the right side of x in array. Elements for which no smaller element exist (on right side), consider next smaller element as -1.

Examples:

- a) For any array, rightmost element always has next smaller element as -1.
- b) For an array which is sorted in increasing order, all elements have next smaller element as -1.
- c) For the input array [4, 8, 5, 2, 25], the next smaller elements for each element are as follows.

Element		NSE
4	-->	2
8	-->	5
5	-->	2
2	-->	-1
25	-->	-1

- d) For the input array [13, 7, 6, 12], the next smaller elements for each element are as follows.

Element		NSE
13	-->	7
7	-->	6
6	-->	-1
12	-->	-1

Method 1 (Simple)

Use two loops: The outer loop picks all the elements one by one. The inner loop looks for the first smaller element for the element picked by outer loop. If a smaller element is found then that element is printed as next, otherwise -1 is printed.

Thanks to Sachin for providing following code.

C

```
// Simple C program to print next smaller elements
// in a given array
#include<stdio.h>

/* prints element and NSE pair for all elements of
arr[] of size n */
void printNSE(int arr[], int n)
{
    int next, i, j;
    for (i=0; i<n; i++)
    {
        next = -1;
        for (j = i+1; j<n; j++)
        {
            if (arr[i] > arr[j])
            {
                next = arr[j];
                break;
            }
        }
        printf("%d -- %d\n", arr[i], next);
    }
}

int main()
{
    int arr[] = {11, 13, 21, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    printNSE(arr, n);
    return 0;
}
```

Java

```
// Simple Java program to print next
// smaller elements in a given array

class Main {
```

```
/* prints element and NSE pair for
all elements of arr[] of size n */
static void printNSE(int arr[], int n)
{
    int next, i, j;
    for (i = 0; i < n; i++) {
        next = -1;
        for (j = i + 1; j < n; j++) {
            if (arr[i] > arr[j]) {
                next = arr[j];
                break;
            }
        }
        System.out.println(arr[i] + " -- " + next);
    }
}

public static void main(String args[])
{
    int arr[] = { 11, 13, 21, 3 };
    int n = arr.length;
    printNSE(arr, n);
}
```

Python

```
# Function to print element and NSE pair for all elements of list
def printNSE(arr):

    for i in range(0, len(arr), 1):

        next = -1
        for j in range(i + 1, len(arr), 1):
            if arr[i] > arr[j]:
                next = arr[j]
                break

        print(str(arr[i]) + " -- " + str(next))

# Driver program to test above function
arr = [11, 13, 21, 3]
printNSE(arr)

# This code is contributed by Sunny Karira
```

C#

```
// Simple C# program to print next
// smaller elements in a given array
using System;

class GFG {

    /* prints element and NSE pair for
    all elements of arr[] of size n */
    static void printNSE(int[] arr, int n)
    {
        int next, i, j;
        for (i = 0; i < n; i++) {
            next = -1;
            for (j = i + 1; j < n; j++) {
                if (arr[i] > arr[j]) {
                    next = arr[j];
                    break;
                }
            }
            Console.WriteLine(arr[i] + " -- " + next);
        }
    }

    // driver code
    public static void Main()
    {
        int[] arr = { 11, 13, 21, 3 };
        int n = arr.Length;

        printNSE(arr, n);
    }
}

// This code is contributed by Sam007
```

PHP

```
<?php
// Simple PHP program to print next
// smaller elements in a given array

/* prints element and NSE pair for
all elements of arr[] of size n */
function printNSE($arr, $n)
{
    for ($i = 0; $i < $n; $i++)
    {
        $next = -1;
```

```

        for ($j = $i + 1; $j < $n; $j++)
        {
            if ($arr[$i] > $arr[$j])
            {
                $next = $arr[$j];
                break;
            }
        }
        echo $arr[$i]. " -- " . $next. "\n";
    }
}

// Driver Code
$arr= array(11, 13, 21, 3);
$n = count($arr);
printNSE($arr, $n);

// This code is contributed by Sam007
?>

```

Output:

```

11 -- 3
13 -- 3
21 -- 3
3 -- -1

```

Time Complexity: $O(n^2)$. The worst case occurs when all elements are sorted in decreasing order.

Method 2 (Using Stack)

This problem is similar to [next greater element](#). Here we maintain items in increasing order in the stack (instead of decreasing in next greater element problem).

- 1) Push the first element to stack.
- 2) Pick rest of the elements one by one and follow following steps in loop.
 -a) Mark the current element as *next*.
 -b) If stack is not empty, then pop an element from stack and compare it with *next*.
 -c) If *next* is smaller than the popped element, then *next* is the next smaller element for the popped element.
 -d) Keep popping from the stack while the popped element is greater than *next*. *next* becomes the next smaller element for all such popped elements
- 3) After the loop in step 2 is over, pop all the elements from stack and print -1 as next element for them.

C++


```

// A Stack based C++ program to find next
// smaller element for all array elements.
#include <bits/stdc++.h>

using namespace std;

/* prints element and NSE pair for all
elements of arr[] of size n */
void printNSE(int arr[], int n)
{
    stack<int> s;

    /* push the first element to stack */
    s.push(arr[0]);

    // iterate for rest of the elements
    for (int i = 1; i < n; i++) {

        if (s.empty()) {
            s.push(arr[i]);
            continue;
        }

        /* if stack is not empty, then
        pop an element from stack.
        If the popped element is smaller
        than next, then
        a) print the pair
        b) keep popping while elements are
        smaller and stack is not empty */
        while (s.empty() == false && s.top() > arr[i]) {
            cout << s.top() << " --> " << arr[i] << endl;
            s.pop();
        }

        /* push next to stack so that we can find
        next smaller for it */
        s.push(arr[i]);
    }

    /* After iterating over the loop, the remaining
    elements in stack do not have the next smaller
    element, so print -1 for them */
    while (s.empty() == false) {
        cout << s.top() << " --> " << -1 << endl;
        s.pop();
    }
}

```

```
/* Driver program to test above functions */
int main()
{
    int arr[] = { 11, 13, 21, 3 };
    int n = sizeof(arr) / sizeof(arr[0]);
    printNSE(arr, n);
    return 0;
}
```

Java

```
// A Stack based Java program to find next
// smaller element for all array elements.
import java.util.*;
import java.lang.*;
import java.io.*;

class GFG
{
    /* prints element and NSE pair for all
    elements of arr[] of size n */
    public static void printNSE(int arr[], int n)
    {
        Stack<Integer> s = new Stack<Integer>();

        /* push the first element to stack */
        s.push(arr[0]);

        // iterate for rest of the elements
        for (int i = 1; i < n; i++) {

            if (s.empty()) {
                s.push(arr[i]);
                continue;
            }

            /* if stack is not empty, then
            pop an element from stack.
            If the popped element is smaller
            than next, then
            a) print the pair
            b) keep popping while elements are
            smaller and stack is not empty */
            while (s.empty() == false && s.peek() > arr[i]) {
                System.out.println(s.peek() + " --> " + arr[i]);
                s.pop();
            }
        }
    }
}
```

```

    }

    /* push next to stack so that we can find
    next smaller for it */
    s.push(arr[i]);
}

/* After iterating over the loop, the remaining
elements in stack do not have the next smaller
element, so print -1 for them */
while (s.empty() == false) {
    System.out.println(s.peek() + " --> " + "-1");
    s.pop();
}
}
/* Driver program to test above functions */
public static void main (String[] args) {
    int arr[] = { 11, 13, 21, 3};
    int n = arr.length;
    printNSE(arr, n);
}
}

```

Output:

```

21 --> 3
13 --> 3
11 --> 3
3 --> -1

```

Time Complexity: $O(n)$. The worst case occurs when all elements are sorted in increasing order. If elements are sorted in increasing order, then every element is processed at most 4 times.

- a) Initially pushed to the stack.
- b) Popped from the stack when next element is being processed.
- c) Pushed back to the stack because next element is smaller.
- d) Popped from the stack in step 3 of algo.

How to get elements in same order as input?

The above approach may not produce output elements in same order as input. To achieve same order, we can use an `unordered_map` in C++ (or `HashMap` in Java).

C++

```

// A Stack based C++ program to find next
// smaller element for all array elements

```

```

// in same order as input.
#include <bits/stdc++.h>
using namespace std;

/* prints element and NSE pair for all
elements of arr[] of size n */
void printNSE(int arr[], int n)
{
    stack<int> s;
    unordered_map<int, int> mp;

    /* push the first element to stack */
    s.push(arr[0]);

    // iterate for rest of the elements
    for (int i = 1; i < n; i++) {

        if (s.empty()) {
            s.push(arr[i]);
            continue;
        }

        /* if stack is not empty, then
        pop an element from stack.
        If the popped element is smaller
        than next, then
        a) print the pair
        b) keep popping while elements are
        smaller and stack is not empty */
        while (s.empty() == false && s.top() > arr[i]) {
            mp[s.top()] = arr[i];
            s.pop();
        }

        /* push next to stack so that we can find
        next smaller for it */
        s.push(arr[i]);
    }

    /* After iterating over the loop, the remaining
    elements in stack do not have the next smaller
    element, so print -1 for them */
    while (s.empty() == false) {
        mp[s.top()] = -1;
        s.pop();
    }
}

```

```
        for (int i=0; i<n; i++)
            cout << arr[i] << " ---> " << mp[arr[i]] << endl;
    }

    /* Driver program to test above functions */
    int main()
    {
        int arr[] = { 11, 13, 21, 3 };
        int n = sizeof(arr) / sizeof(arr[0]);
        printNSE(arr, n);
        return 0;
    }
```

Java

```
// A Stack based Java program to find next
// smaller element for all array elements
// in same order as input.
import java.util.*;
import java.lang.*;
import java.io.*;

class GFG
{
    /* prints element and NSE pair for all
    elements of arr[] of size n */
    public static void printNSE(int arr[], int n)
    {
        Stack<Integer> s = new Stack<Integer>();
        HashMap<Integer,Integer> mp = new HashMap<Integer,Integer>();

        /* push the first element to stack */
        s.push(arr[0]);

        // iterate for rest of the elements
        for (int i = 1; i < n; i++) {

            if (s.empty()) {
                s.push(arr[i]);
                continue;
            }

            /* if stack is not empty, then
            pop an element from stack.
            If the popped element is smaller
            than next, then
```

```
a) print the pair
b) keep popping while elements are
smaller and stack is not empty */

while (s.empty() == false && s.peek() > arr[i]) {
    mp.put(s.peek(), arr[i]);
    s.pop();
}

/* push next to stack so that we can find
next smaller for it */
s.push(arr[i]);
}

/* After iterating over the loop, the remaining
elements in stack do not have the next smaller
element, so print -1 for them */
while (s.empty() == false) {
    mp.put(s.peek(), -1);
    s.pop();
}

for (int i=0; i<n; i++)
    System.out.println(arr[i] + " ---> " + mp.get(arr[i]));
}

/* Driver program to test above functions */
public static void main (String[] args) {
    int arr[] = { 11, 13, 21, 3};
    int n = arr.length;
    printNSE(arr, n);
}
}
```

Output:

```
11 ---> 3
13 ---> 3
21 ---> 3
3 ---> -1
```

Improved By : [nabaneet247](#)

Source

<https://www.geeksforgeeks.org/next-smaller-element/>

Chapter 78

Number of NGEs to the right

Number of NGEs to the right - GeeksforGeeks

Given an array of n integers and q queries, print the number of [next greater elements](#) to the right of the given index element.

Examples:

```
Input : a[] = {3, 4, 2, 7, 5, 8, 10, 6}
        q = 2
        index = 0,
        index = 5
```

```
Output: 4
        1
```

Explanation: the next greater elements to the right of 3(index 0) are 4, 7, 8, 10. The next greater elements to the right of 8(index 5) are 10.

A **naive approach** is to iterate for every query from index to end, and find out the number of next greater elements to the right. This won't be efficient enough as we run two nested loops .

Time complexity: $O(n)$ to answer a query.

Auxiliary space: $O(1)$

Better approach is to store the next greater index of every element and run a loop for every query that iterates from index and keeping the increasing counter as $j = \text{next}[i]$. This will avoid checking all elements and will directly jump to the next greater element of every element. But this won't be efficient enough in cases like 1 2 3 4 5 6, where the next greater elements are sequentially increasing, ending it up in taking $O(n)$ for every query.

Time complexity : $O(n)$ to answer a query.

Auxiliary space : $O(n)$ for next greater element.

Efficient approach is to store the next greater elements index using next greater element in a `next[]` array. Then create a `dp[]` array that starts from `n-2`, as `n-1`th index will have no elements to its right and `dp[n-1] = 0`. While traversing from back we use dynamic programming to count the number of elements to the right where we use memoization as `dp[next[i]]` which gives us a count of the numbers to the right of the next greater element of the current element, hence we add 1 to it. If `next[i]=-1` then we do not have any element to the right hence `dp[i]=0`. `dp[index]` stores the count of the number of next greater elements to the right.

Below is the c++ implementation of the above approach

```
#include <bits/stdc++.h>
using namespace std;

// array to store the next greater element index
void fillNext(int next[], int a[], int n)
{
    // use of stl stack in c++
    stack<int> s;

    // push the 0th index to the stack
    s.push(0);

    // traverse in the loop from 1-nth index
    for (int i = 1; i < n; i++) {

        // iterate till loop is empty
        while (!s.empty()) {

            // get the topmost index in the stack
            int cur = s.top();

            // if the current element is greater
            // then the top index-th element, then
            // this will be the next greatest index
            // of the top index-th element
            if (a[cur] < a[i]) {

                // initialize the cur index position's
                // next greatest as index
                next[cur] = i;

                // pop the cur index as its greater
                // element has been found
                s.pop();
            }

            // if not greater then break
            else
```

```
        break;
    }

    // push the i index so that its next greatest
    // can be found
    s.push(i);
}

// iterate for all other index left inside stack
while (!s.empty()) {

    int cur = s.top();

    // mark it as -1 as no element in greater
    // then it in right
    next[cur] = -1;

    s.pop();
}

// function to count the number of next greater numbers to the right
void count(int a[], int dp[], int n)
{
    // initializes the next array as 0
    int next[n];
    memset(next, 0, sizeof(next));

    // calls the function to pre-calculate
    // the next greatest element indexes
    fillNext(next, a, n);

    for (int i = n - 2; i >= 0; i--) {

        // if the i-th element has no next
        // greater element to right
        if (next[i] == -1)
            dp[i] = 0;

        // Count of next greater numbers to right.
        else
            dp[i] = 1 + dp[next[i]];
    }
}

// answers all queries in O(1)
int answerQuery(int dp[], int index)
{
}
```

```
// returns the number of next greater
// elements to the right.
return dp[index];
}

// driver program to test the above function
int main()
{
    int a[] = { 3, 4, 2, 7, 5, 8, 10, 6 };
    int n = sizeof(a) / sizeof(a[0]);

    int dp[n];

    // calls the function to count the number
    // of greater elements to the right for
    // every element.
    count(a, dp, n);

    // query 1 answered
    cout << answerQuery(dp, 3) << endl;

    // query 2 answered
    cout << answerQuery(dp, 6) << endl;

    // query 3 answered
    cout << answerQuery(dp, 1) << endl;

    return 0;
}
```

Output:

```
2
0
3
```

Time complexity: $O(1)$ to answer a query.
Auxiliary Space: $O(n)$

Source

<https://www.geeksforgeeks.org/number-nges-right/>

Chapter 79

Pattern Occurrences : Stack Implementation Java

Pattern Occurrences : Stack Implementation Java - GeeksforGeeks

Suppose we have two Strings :- Pattern and Text

pattern: consisting of unique characters

text: consisting of any length

We need to find the number of **patterns** that can be obtained from **text** removing each and every occurrence of Pattern in the Text.

Example:

Input :

Pattern : ABC

Text : ABABCABCC

Output :

3

Occurrences found at:

4 7 8

Explanation

Occurrences and their removal in the order

1. ABABCABCC

2. ABABCC

3. ABC

The idea is to use stack data structure.

1. Initialize a **pointer** to beginning for matching the occurrences in the pattern with 0 and **counter** to 0.

2. Check if pattern and text have same character at the present index.

3. If the pointer is to the end of pattern that means all the previous characters have been

found in an **increasing subsequential order** increment the **counter** by 1.

4. If not, keep incrementing the **pointer** by 1 if characters are same.

5. If the characters are different in both the strings, check if the character is same as the first character of the pattern (i.e. `pointer = 0`).

6. If yes, add the remaining characters from the present pointer to length of the pattern to a stack and check if they are present in order that the pattern can be formed from the stack. Also, initialize the pointer now to 1 because we already had checked for `pointer = 0` (in **step 5**).

7. If matches, empty the stack to **null**. Else, remove the first character and keep adding the rest of the substring for checking for further of the steps.

8. If any added String to the Stack matches the pattern **increment counter by 1** and initialize **pointer by 0**.

9. Repeat all these steps **for all the indexes** of the text length.

10. Print the counter and occurrences.

11. Basic task of Stack is **handling the pending operations** that might be possible occurrences.

Example Explanation according to above algorithm:

TEXT: ABABCABCC

PATTERN: ABC

`pointer = 0`

`counter = 0`

A B A B C A B C C

0 1 2 3 4 5 6 7 8

at index = 0

`pointer = 0`

`stack = []`

at index = 1

`pointer = 1`

`stack = []`

at index = 2

`pointer = 0`

`stack = ['C']`

at index = 3

`pointer = 1`

`stack = ['C']`

at index = 4

`pointer = 2`

`counter += 1`

`pointer = 0`

`stack = ['C']`

same for index 5,6,7 according to above method

```
at index = 8
pop from Stack
counter += 1
clear Stack
```

Code in Java for the algorithm:

Prerequisite : [Stack class in Java](#)

```
import java.util.ArrayList;
import java.util.Stack;

class StackImplementation
{
    // custom class for returning multiple values
    class Data
    {
        ArrayList<Integer> present;
        int count;

        public Data(ArrayList<Integer> present, int count)
        {
            this.present = present;
            this.count = count;
        }
    }
    public Data Solution(char pattern[], char text[])
    {
        // stores the indices for all occurrences
        ArrayList<Integer> list = new ArrayList<>();
        Stack<String> stack = new Stack<>();

        // present index pointer searched for in
        // the entire array of string characters
        int p = 0;

        //count of all the number of occurrences
        int counter = 0 ;

        // any random number less than 0 to mark
        // the previous index where the occurrence
        // was found
        int lastOccurrence = -10;

        // traversing all the indexes of the text
        // searching for possible pattern
        for (int i = 0; i < text.length; i ++)
```

```
{
    // if the present index and the pointer in
    // the pattern is at same character
    if(text[i] == pattern[p])
    {
        // and if that character is the end of
        // the pattern to be found
        if(text[i] == pattern[pattern.length - 1])
        {
            //index at which pattern is found
            list.add(i);

            // incrementing total occurrences by 1
            counter ++;

            // last found index to be initialized
            // to present index
            lastOccurrence = i;

            // begin the search for the next pointer
            // again from 0th index of the pattern
            p = 0;
        }
        else
        {
            // if present character at pattern and index
            // is same but still not the end of pattern
            p ++;
        }
    }

    // if characters are not same
    else
    {
        // if the present character is same as the 1st
        // character of the pattern
        // here 0 = pointer in the pattern fixed to 0
        if(text[i] == pattern[0])
        {
            // assume a temporary string
            String temp = "";

            // and add all characters to it to the pattern
            // length from the present pointer to the end
            for (int i1 = p; i1 < pattern.length; i1 ++ )
                temp += pattern[i1];

            // push the present pattern length into the stack
```

```
// for checking if pattern is same as subsequence
// of the text
stack.push(temp);

//pattern at pointer = 0 already checked so we
// start from 1 for the next step
p = 1;
}
else
{
    // if the previous occurrence was just before
    // the present index
    if (lastOccurrence == i - 1)
    {
        // if the stack is empty place the pointer = 0
        if (stack.isEmpty())
            p = 0;
        else
        {
            // pick up the present possible pattern
            String temp = stack.pop();

            // check if it's character has the matching
            // occurrence
            if (temp.charAt(0) == text[i])
            {
                //increment last index by the present index
                // so that net index is checked
                lastOccurrence = i;

                // check if stack character is last character
                // in the pattern
                if (temp.charAt(0) == pattern[pattern.length - 1])
                {
                    // index found
                    list.add(i);

                    // increment occurrences by 1
                    counter ++;
                }
            }
            else
            {
                // if present index character doesn't
                // match the last character in the pattern
                // remove the first character which was same
                // and check for further occurrences of the
                // remaining letters in the stack string
                temp = temp.substring(1, temp.length());
            }
        }
    }
}
```



```
        // add the remaining string back to stack
        // for further review
        stack.push(temp);
    }
}
// if first string character in the stack doesn't
// match the present character in the text
else
{
    // if stack is not empty empty it.
    if (!stack.isEmpty())
        stack.clear();

    // reinitialize the pointer back to 0 for
    // checking pattern from beginning
    p = 0;
}
}
}
else
{
    // empty the stack under any other circumstances
    if (!stack.isEmpty())
        stack.clear();

    // reinitialize the pointer back to 0 for
    // checking pattern from beginning
    p = 0;
}
}
}
// return the result
return new Data(list, counter);
}

public static void main(String args[])
{
    // the simple pattern to be matched
    char[] pattern = "ABC".toCharArray();

    // the input string in which the number of
    // occurrences can be found out after removing
    // each occurrence.
    char[] text = "ABABCABCC".toCharArray();

    StackImplementation obj = new StackImplementation();
```

```
Data data = obj.Solution(pattern, text);

int count = data.count;
ArrayList<Integer> list = data.present;
System.out.println(count);
if (count > 0)
{
    System.out.println("Occurrences found at:");
    for (int i : list)
        System.out.print(i + " ");
    }
}
```

Output:

```
3
Occurrences found at:
4 7 8
```

References:

1. [Stack Java Documentation.](#)
2. [Custom ArrayList and Class in Java.](#)
3. [More on Stack Operations.](#)

Source

<https://www.geeksforgeeks.org/pattern-occurrences-stack-implementation-java/>

Chapter 80

Postfix to Infix

Postfix to Infix - GeeksforGeeks

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

Postfix expression: The expression of the form $a \text{ b op}$. When an operator is followed for every pair of operands.

Postfix notation, also known as reverse Polish notation, is a syntax for mathematical expressions in which the mathematical operator is always placed after the operands. Though postfix expressions are easily and efficiently evaluated by computers, they can be difficult for humans to read. Complex expressions using standard parenthesized infix notation are often more readable than the corresponding postfix expressions. Consequently, we would sometimes like to allow end users to work with infix notation and then convert it to postfix notation for computer processing. Sometimes, moreover, expressions are stored or generated in postfix, and we would like to convert them to infix for the purpose of reading and editing

Examples:

Input : $abc++$
Output : $(a + (b + c))$

Input : $ab*c+$
Output : $((a*b)+c)$

We have already discussed [Infix to Postfix](#). Below is algorithm for Postfix to Infix.

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.

```
// CPP program to find infix for
// a given postfix.
#include <bits/stdc++.h>
using namespace std;

bool isOperand(char x)
{
    return (x >= 'a' && x <= 'z') ||
           (x >= 'A' && x <= 'Z');
}

// Get Infix for a given postfix
// expression
string getInfix(string exp)
{
    stack<string> s;

    for (int i=0; exp[i]!='\0'; i++)
    {
        // Push operands
        if (isOperand(exp[i]))
        {
            string op(1, exp[i]);
            s.push(op);
        }

        // We assume that input is
        // a valid postfix and expect
        // an operator.
        else
        {
            string op1 = s.top();
            s.pop();
            string op2 = s.top();
            s.pop();
            s.push("(" + op2 + exp[i] +
                  op1 + ")");
        }
    }

    // There must be a single element
```

```
        // in stack now which is the required
        // infix.
        return s.top();
    }

    // Driver code
    int main()
    {
        string exp = "ab*c+";
        cout << getInfix(exp);
        return 0;
    }
```

Output:

((a*b)+c)

Source

<https://www.geeksforgeeks.org/postfix-to-infix/>

Chapter 81

Postfix to Prefix Conversion

Postfix to Prefix Conversion - GeeksforGeeks

Postfix: An expression is called the postfix expression if the operator appears in the expression after the operands. Simply of the form (operand1 operand2 operator).

Example : $AB+CD-*$ (Infix : $(A+B) * (C-D)$)

Prefix : An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2).

Example : $*+AB-CD$ (Infix : $(A+B) * (C-D)$)

Given a Postfix expression, convert it into a Prefix expression.

Conversion of Postfix expression directly to Prefix without going through the process of converting them first to Infix and then to Prefix is much better in terms of computation and better understanding the expression (Computers evaluate using Postfix expression).

Examples:

Input : Postfix : $AB+CD-*$

Output : Prefix : $*+AB-CD$

Explanation : Postfix to Infix : $(A+B) * (C-D)$

Infix to Prefix : $*+AB-CD$

Input : Postfix : $ABC/-AK/L-*$

Output : Prefix : $*-A/BC-/AKL$

Explanation : Postfix to Infix : $A-(B/C)*(A/K)-L$

Infix to Prefix : $*-A/BC-/AKL$

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.

```
// CPP Program to convert postfix to prefix
#include <iostream>
#include <stack>
using namespace std;

// function to check if character is operator or not
bool isOperator(char x) {
    switch (x) {
        case '+':
        case '-':
        case '/':
        case '*':
            return true;
    }
    return false;
}

// Convert postfix to Prefix expression
string postToPre(string post_exp) {
    stack<string> s;

    // length of expression
    int length = post_exp.size();

    // reading from right to left
    for (int i = 0; i < length; i++) {

        // check if symbol is operator
        if (isOperator(post_exp[i])) {

            // pop two operands from stack
            string op1 = s.top();
            s.pop();
            string op2 = s.top();
            s.pop();

            // concat the operands and operator
            string temp = post_exp[i] + op2 + op1;

            // Push string temp back to stack
            s.push(temp);
        }
    }
}
```

```
    }

    // if symbol is an operand
    else {

        // push the operand to the stack
        s.push(string(1, post_exp[i]));
    }
}

// stack[0] contains the Prefix expression
return s.top();
}

// Driver Code
int main() {
    string post_exp = "ABC/-AK/L-*";
    cout << "Prefix : " << postToPre(post_exp);
    return 0;
}
```

Output:

Prefix : *-A/BC-/AKL

Source

<https://www.geeksforgeeks.org/postfix-prefix-conversion/>

Chapter 82

Prefix to Infix Conversion

Prefix to Infix Conversion - GeeksforGeeks

Infix : An expression is called the Infix expression if the operator appears in between the operands in the expression. Simply of the form (operand1 operator operand2).

Example : (A+B) * (C-D)

Prefix : An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2).

Example : *+AB-CD (Infix : (A+B) * (C-D))

Given a Prefix expression, convert it into a Infix expression.

Computers usually does the computation in either prefix or postfix (usually postfix). But for humans, its easier to understand an Infix expression rather than a prefix. Hence conversion is need for human understanding.

Examples:

Input : Prefix : *+AB-CD

Output : Infix : ((A+B)*(C-D))

Input : Prefix : *-A/BC-/AKL

Output : Infix : ((A-(B/C))*((A/K)-L))

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.

```
// CPP Program to convert prefix to Infix
#include <iostream>
#include <stack>
using namespace std;

// function to check if character is operator or not
bool isOperator(char x) {
    switch (x) {
        case '+':
        case '-':
        case '/':
        case '*':
            return true;
    }
    return false;
}

// Convert prefix to Infix expression
string preToInfix(string pre_exp) {
    stack<string> s;

    // length of expression
    int length = pre_exp.size();

    // reading from right to left
    for (int i = length - 1; i >= 0; i--) {

        // check if symbol is operator
        if (isOperator(pre_exp[i])) {

            // pop two operands from stack
            string op1 = s.top();    s.pop();
            string op2 = s.top();    s.pop();

            // concat the operands and operator
            string temp = "(" + op1 + pre_exp[i] + op2 + ")";

            // Push string temp back to stack
            s.push(temp);
        }

        // if symbol is an operand
        else {

            // push the operand to the stack
            s.push(string(1, pre_exp[i]));
        }
    }
}
```

```
    // Stack now contains the Infix expression
    return s.top();
}

// Driver Code
int main() {
    string pre_exp = "*-A/BC-/AKL";
    cout << "Infix : " << preToInfix(pre_exp);
    return 0;
}
```

Output:

Infix : ((A-(B/C))*((A/K)-L))

Source

<https://www.geeksforgeeks.org/prefix-infix-conversion/>

Chapter 83

Prefix to Postfix Conversion

Prefix to Postfix Conversion - GeeksforGeeks

Prefix : An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2).

Example : $*+AB-CD$ (Infix : $(A+B) * (C-D)$)

Postfix: An expression is called the postfix expression if the operator appears in the expression after the operands. Simply of the form (operand1 operand2 operator).

Example : $AB+CD-*$ (Infix : $(A+B * (C-D))$

Given a Prefix expression, convert it into a Postfix expression.

Conversion of Prefix expression directly to Postfix without going through the process of converting them first to Infix and then to Postfix is much better in terms of computation and better understanding the expression (Computers evaluate using Postfix expression).

Examples:

Input : Prefix : $*+AB-CD$
Output : Postfix : $AB+CD-*$
Explanation : Prefix to Infix : $(A+B) * (C-D)$
Infix to Postfix : $AB+CD-*$

Input : Prefix : $*-A/BC-/AKL$
Output : Postfix : $ABC/-AK/L-*$
Explanation : Prefix to Infix : $A-(B/C)*(A/K)-L$
Infix to Postfix : $ABC/-AK/L-*$

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.

```
// CPP Program to convert prefix to postfix
#include <iostream>
#include <stack>
using namespace std;

// function to check if character is operator or not
bool isOperator(char x) {
    switch (x) {
        case '+':
        case '-':
        case '/':
        case '*':
            return true;
    }
    return false;
}

// Convert prefix to Postfix expression
string preToPost(string pre_exp) {

    stack<string> s;

    // length of expression
    int length = pre_exp.size();

    // reading from right to left
    for (int i = length - 1; i >= 0; i--) {

        // check if symbol is operator
        if (isOperator(pre_exp[i])) {

            // pop two operands from stack
            string op1 = s.top(); s.pop();
            string op2 = s.top(); s.pop();

            // concat the operands and operator
            string temp = op1 + op2 + pre_exp[i];

            // Push string temp back to stack
            s.push(temp);
        }
    }
}
```

```
// if symbol is an operand
else {

    // push the operand to the stack
    s.push(string(1, pre_exp[i]));
}

// stack contains only the Postfix expression
return s.top();
}

// Driver Code
int main() {
    string pre_exp = "*-A/BC-/AKL";
    cout << "Postfix : " << preToPost(pre_exp);
    return 0;
}
```

Output:

Postfix : ABC/-AK/L-*

Source

<https://www.geeksforgeeks.org/prefix-postfix-conversion/>

Chapter 84

Preorder from Inorder and Postorder traversals

Preorder from Inorder and Postorder traversals - GeeksforGeeks

Given Inorder and Postorder traversals of a binary tree, print Preorder traversal.

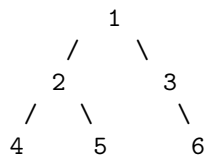
Example:

Input: Postorder traversal post[] = {4, 5, 2, 6, 3, 1}

Inorder traversal in[] = {4, 2, 5, 1, 3, 6}

Output: Preorder traversal 1, 2, 4, 5, 3, 6

Trversals in the above example represents following tree



A **naive method** is to first [construct the tree from given postorder and inorder](#), then use simple recursive method to print preorder traversal of the constructed tree.

InOrder(root) visits nodes in the following order:

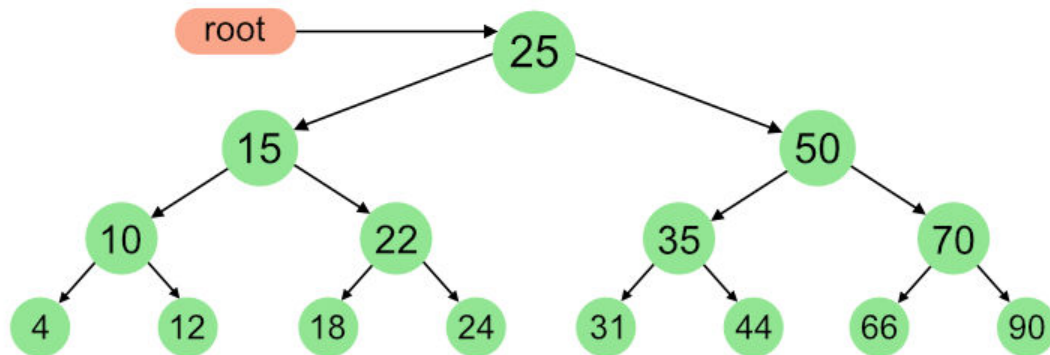
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



We can print preorder traversal without constructing the tree. The idea is, root is always the first item in preorder traversal and it must be the last item in postorder traversal. We first push right subtree to a stack, then left subtree and finally we push root. Finally we print contents of stack. To find boundaries of left and right subtrees in `post[]` and `in[]`, we search root in `in[]`, all elements before root in `in[]` are elements of left subtree and all elements after root are elements of right subtree. In `post[]`, all elements after index of root in `in[]` are elements of right subtree. And elements before index (including the element at index and excluding the first element) are elements of left subtree.

```
// Java program to print Postorder traversal from given
// Inorder and Preorder traversals.
import java.util.Stack;

public class PrintPre {

    static int postIndex;

    // Fills preorder traversal of tree with given
    // inorder and postorder traversals in a stack
    void fillPre(int[] in, int[] post, int inStrt,
```



```
        int inEnd, Stack<Integer> s)
{
    if (inStrt > inEnd)
        return;

    // Find index of next item in postorder traversal in
    // inorder.
    int val = post[postIndex];
    int inIndex = search(in, val);
    postIndex--;

    // traverse right tree
    fillPre(in, post, inIndex + 1, inEnd, s);

    // traverse left tree
    fillPre(in, post, inStrt, inIndex - 1, s);

    s.push(val);
}

// This function basically initializes postIndex
// as last element index, then fills stack with
// reverse preorder traversal using printPre
void printPreMain(int[] in, int[] post)
{
    int len = in.length;
    postIndex = len - 1;
    Stack<Integer> s = new Stack<Integer>();
    fillPre(in, post, 0, len - 1, s);
    while (s.empty() == false)
        System.out.print(s.pop() + " ");
}

// A utility function to search data in in[]
int search(int[] in, int data)
{
    int i = 0;
    for (i = 0; i < in.length; i++)
        if (in[i] == data)
            return i;
    return i;
}

// Driver code
public static void main(String ars[])
{
    int in[] = { 4, 10, 12, 15, 18, 22, 24, 25,
                 31, 35, 44, 50, 66, 70, 90 };
}
```

```
        int post[] = { 4, 12, 10, 18, 24, 22, 15, 31,
                       44, 35, 66, 90, 70, 50, 25 };
        PrintPre tree = new PrintPre();
        tree.printPreMain(in, post);
    }
}
```

Output:

25 15 10 4 12 22 18 24 50 35 31 44 70 66 90

Time Complexity: The above function visits every node in array. For every visit, it calls search which takes $O(n)$ time. Therefore, overall time complexity of the function is $O(n^2)$

$O(n)$ Solution

We can further optimize above solution to first hash all items of inorder traversal so that we do not have to linearly search items. With hash table available to us, we can search an item in $O(1)$ time.

```
// Java program to print Postorder traversal from given
// Inorder and Preorder traversals.
import java.util.Stack;
import java.util.HashMap;

public class PrintPre {

    static int postIndex;

    // Fills preorder traversal of tree with given
    // inorder and postorder traversals in a stack
    void fillPre(int[] in, int[] post, int inStrt, int inEnd,
                Stack<Integer> s, HashMap<Integer, Integer> hm)
    {
        if (inStrt > inEnd)
            return;

        // Find index of next item in postorder traversal in
        // inorder.
        int val = post[postIndex];
        int inIndex = hm.get(val);
        postIndex--;

        // traverse right tree
        fillPre(in, post, inIndex + 1, inEnd, s, hm);

        // traverse left tree
```

```
        fillPre(in, post, inStrt, inIndex - 1, s, hm);

        s.push(val);
    }

    // This function basically initializes postIndex
    // as last element index, then fills stack with
    // reverse preorder traversal using printPre
    void printPreMain(int[] in, int[] post)
    {
        int len = in.length;
        postIndex = len - 1;
        Stack<Integer> s = new Stack<Integer>();

        // Insert values in a hash map and their indexes.
        HashMap<Integer, Integer> hm =
            new HashMap<Integer, Integer>();
        for (int i = 0; i < in.length; i++)
            hm.put(in[i], i);

        // Fill preorder traversal in a stack
        fillPre(in, post, 0, len - 1, s, hm);

        // Print contents of stack
        while (s.empty() == false)
            System.out.print(s.pop() + " ");
    }

    // Driver code
    public static void main(String ars[])
    {
        int in[] = { 4, 10, 12, 15, 18, 22, 24, 25,
                     31, 35, 44, 50, 66, 70, 90 };
        int post[] = { 4, 12, 10, 18, 24, 22, 15, 31,
                      44, 35, 66, 90, 70, 50, 25 };
        PrintPre tree = new PrintPre();
        tree.printPreMain(in, post);
    }
}
```

Output:

25 15 10 4 12 22 18 24 50 35 31 44 70 66 90

Time Complexity: $O(n)$

Source

<https://www.geeksforgeeks.org/preorder-from-inorder-and-postorder-traversals/>

Chapter 85

Previous greater element

Previous greater element - GeeksforGeeks

Given an array of distinct elements, find previous greater element for every element. If previous greater element does not exist, print -1.

Examples:

Input : arr[] = {10, 4, 2, 20, 40, 12, 30}
Output : -1, 10, 4, -1, -1, 40, 40

Input : arr[] = {10, 20, 30, 40}
Output : -1, -1, -1, -1

Input : arr[] = {40, 30, 20, 10}
Output : -1, 40, 30, 20

Expected time complexity : $O(n)$

A **simple solution** is to run two nested loops. The outer loop picks an element one by one. The inner loop, find the previous element that is greater.

C++

```
// C++ program previous greater element
// A naive solution to print previous greater
// element for every element in an array.
#include <bits/stdc++.h>
using namespace std;

void prevGreater(int arr[], int n)
{
    // Previous greater for first element never
```

```
// exists, so we print -1.
cout << "-1, ";

// Let us process remaining elements.
for (int i = 1; i < n; i++) {

    // Find first element on left side
    // that is greater than arr[i].
    int j;
    for (j = i-1; j >= 0; j--) {
        if (arr[i] < arr[j]) {
            cout << arr[j] << ", ";
            break;
        }
    }

    // If all elements on left are smaller.
    if (j == -1)
        cout << "-1, ";
    }
}

// Driver code
int main()
{
    int arr[] = { 10, 4, 2, 20, 40, 12, 30 };
    int n = sizeof(arr) / sizeof(arr[0]);
    prevGreater(arr, n);
    return 0;
}
```

Java

```
// Java program previous greater element
// A naive solution to print
// previous greater element
// for every element in an array.
import java.io.*;
import java.util.*;
import java.lang.*;

class GFG
{
    static void prevGreater(int arr[],
                           int n)
    {
        // Previous greater for
        // first element never
        // exists, so we print -1.
    }
}
```

```
System.out.print("-1, ");

// Let us process
// remaining elements.
for (int i = 1; i < n; i++)
{
    // Find first element on
    // left side that is
    // greater than arr[i].
    int j;
    for (j = i-1; j >= 0; j--)
    {
        if (arr[i] < arr[j])
        {
            System.out.print(arr[j] + ", ");
            break;
        }
    }

    // If all elements on
    // left are smaller.
    if (j == -1)
        System.out.print("-1, ");
}

// Driver Code
public static void main(String[] args)
{
    int arr[] = {10, 4, 2, 20, 40, 12, 30};
    int n = arr.length;
    prevGreater(arr, n);
}
}
```

Python 3

```
# Python 3 program previous greater element
# A naive solution to print previous greater
# element for every element in an array.
def prevGreater(arr, n) :

    # Previous greater for first element never
    # exists, so we print -1.
    print("-1",end = ", ")

    # Let us process remaining elements.
```

```
for i in range(1, n) :
    flag = 0

    # Find first element on left side
    # that is greater than arr[i].
    for j in range(i-1, -1, -1) :
        if arr[i] < arr[j] :
            print(arr[j],end = ", ")
            flag = 1
            break

    # If all elements on left are smaller.
    if j == 0 and flag == 0:
        print("-1",end = ", ")

# Driver code
if __name__ == "__main__" :
    arr = [10, 4, 2, 20, 40, 12, 30]
    n = len(arr)
    prevGreater(arr, n)

# This code is contributed by ANKITRAI1
```

PHP

```
<?php
// php program previous greater element
// A naive solution to print previous greater
// element for every element in an array.

function prevGreater(&$arr,$n)
{
    // Previous greater for first element never
    // exists, so we print -1.
    echo( "-1, ");

    // Let us process remaining elements.
    for ($i = 1; $i < $n; $i++)
    {
        // Find first element on left side
        // that is greater than arr[i].
        for ($j = $i-1; $j >= 0; $j--)
        {
            if ($arr[$i] < $arr[$j])
            {
                echo($arr[$j]);
            }
        }
    }
}
```



```
        echo( " ", " ");
        break;
    }
}

// If all elements on left are smaller.
if ($j == -1)
    echo("-1, ");
}
}

// Driver code
$arr = array(10, 4, 2, 20, 40, 12, 30);
$n = sizeof($arr) ;
prevGreater($arr, $n);

//This code is contributed by Shivi_Aggarwal.

?>
```

Output:

-1, 10, 4, -1, -1, 40, 40

An **efficient solution** is to use [stack data structure](#). If we take a closer look, we can notice that this problem is a variation of [stock span problem](#). We maintain previous greater element in a stack.

C++

```
// C++ program previous greater element
// An efficient solution to print previous greater
// element for every element in an array.
#include <bits/stdc++.h>
using namespace std;

void prevGreater(int arr[], int n)
{
    // Create a stack and push index of first element
    // to it
    stack<int> s;
    s.push(arr[0]);

    // Previous greater for first element is always -1.
    cout << "-1, ";

    // Traverse remaining elements
```

```
for (int i = 1; i < n; i++) {

    // Pop elements from stack while stack is not empty
    // and top of stack is smaller than arr[i]. We
    // always have elements in decreasing order in a
    // stack.
    while (s.empty() == false && s.top() < arr[i])
        s.pop();

    // If stack becomes empty, then no element is greater
    // on left side. Else top of stack is previous
    // greater.
    s.empty() ? cout << "-1, " : cout << s.top() << ", ";

    s.push(arr[i]);
}
}
// Driver code
int main()
{
    int arr[] = { 10, 4, 2, 20, 40, 12, 30 };
    int n = sizeof(arr) / sizeof(arr[0]);
    prevGreater(arr, n);
    return 0;
}
```

Java

```
// Java program previous greater element
// An efficient solution to
// print previous greater
// element for every element
// in an array.
import java.io.*;
import java.util.*;
import java.lang.*;

class GFG
{
    static void prevGreater(int arr[],
                           int n)
    {
        // Create a stack and push
        // index of first element
        // to it
        Stack<Integer> s = new Stack<Integer>();
        s.push(arr[0]);
```

```
// Previous greater for
// first element is always -1.
System.out.print("-1, ");

// Traverse remaining elements
for (int i = 1; i < n; i++)
{
    // Pop elements from stack
    // while stack is not empty
    // and top of stack is smaller
    // than arr[i]. We always have
    // elements in decreasing order
    // in a stack.
    while (s.empty() == false &&
           s.peek() < arr[i])
        s.pop();

    // If stack becomes empty, then
    // no element is greater on left
    // side. Else top of stack is
    // previous greater.
    if (s.empty() == true)
        System.out.print("-1, ");
    else
        System.out.print(s.peek() + ", ");

    s.push(arr[i]);
}

// Driver Code
public static void main(String[] args)
{
    int arr[] = { 10, 4, 2, 20, 40, 12, 30 };
    int n = arr.length;
    prevGreater(arr, n);
}
}
```

Output:

-1, 10, 4, -1, -1, 40, 40

Time Complexity: $O(n)$. It seems more than $O(n)$ at first look. If we take a closer look, we can observe that every element of array is added and removed from stack at most once.

So there are total $2n$ operations at most. Assuming that a stack operation takes $O(1)$ time, we can say that the time complexity is $O(n)$.

Auxiliary Space: $O(n)$ in worst case when all elements are sorted in decreasing order.

Improved By : [Shivi_Aggarwal](#), [ANKITRAI1](#)

Source

<https://www.geeksforgeeks.org/previous-greater-element/>

Chapter 86

Print Bracket Number

Print Bracket Number - GeeksforGeeks

Given an expression **exp** of length **n** consisting of some brackets. The task is to print the bracket numbers when the expression is being parsed.

Examples :

Input : (a+(b*c))+(d/e)

Output : 1 2 2 1 3 3

The highlighted brackets in the given expression
(a+(b*c))+(d/e) has been assigned the numbers as:
1 2 2 1 3 3.

Input : (((()))(()))

Output : 1 2 3 3 2 4 5 5 4 1

Source: [Flipkart Interview Experience | Set 49](#).

Approach :

1. Define a variable **left_bnum** = 1.
2. Create a stack **right_bnum**.
3. Now, for i = 0 to n-1.
 - (a) If exp[i] == '(', then print **left_bnum**, push **left_bnum** on to the stack **right_bnum** and finally increment **left_bnum** by 1.
 - (b) Else if exp[i] == ')', then print the top element of the stack **right_bnum** and then pop the top element from the stack.

C++

```
// C++ implementation to print the bracket number
#include <bits/stdc++.h>

using namespace std;

// function to print the bracket number
void printBracketNumber(string exp, int n)
{
    // used to print the bracket number
    // for the left bracket
    int left_bnum = 1;

    // used to obtain the bracket number
    // for the right bracket
    stack<int> right_bnum;

    // traverse the given expression 'exp'
    for (int i = 0; i < n; i++) {

        // if current character is a left bracket
        if (exp[i] == '(') {
            // print 'left_bnum',
            cout << left_bnum << " ";

            // push 'left_bum' on to the stack 'right_bnum'
            right_bnum.push(left_bnum);

            // increment 'left_bnum' by 1
            left_bnum++;
        }

        // else if current character is a right bracket
        else if (exp[i] == ')') {

            // print the top element of stack 'right_bnum'
            // it will be the right bracket number
            cout << right_bnum.top() << " ";

            // pop the top element from the stack
            right_bnum.pop();
        }
    }
}

// Driver program to test above
int main()
{
    string exp = "(a+(b*c))+(d/e)";
```

```
    int n = exp.size();

    printBracketNumber(exp, n);

    return 0;
}
```

Java

```
// Java implementation to
// print the bracket number
import java.io.*;
import java.util.*;

class GFG
{
    // function to print
    // the bracket number
    static void printBracketNumber(String exp,
                                    int n)
    {
        // used to print the
        // bracket number for
        // the left bracket
        int left_bnum = 1;

        // used to obtain the
        // bracket number for
        // the right bracket
        Stack<Integer> right_bnum =
            new Stack<Integer>();

        // traverse the given
        // expression 'exp'
        for (int i = 0; i < n; i++)
        {

            // if current character
            // is a left bracket
            if (exp.charAt(i) == '(')
            {

                // print 'left_bnum',
                System.out.print(
                    left_bnum + " ");

                // push 'left_bum' on to
                // the stack 'right_bnum'
            }
        }
    }
}
```

```
        right_bnum.push(left_bnum);

        // increment 'left_bnum' by 1
        left_bnum++;
    }

    // else if current character
    // is a right bracket
    else if(exp.charAt(i) == ')')
    {

        // print the top element
        // of stack 'right_bnum'
        // it will be the right
        // bracket number
        System.out.print(
            right_bnum.peek() + " ");

        // pop the top element
        // from the stack
        right_bnum.pop();
    }
}

// Driver Code
public static void main(String args[])
{
    String exp = "(a+(b*c))+(d/e)";
    int n = exp.length();

    printBracketNumber(exp, n);
}

// This code is contributed
// by Manish Shaw(manishshaw1)
```

C#

```
// C# implementation to
// print the bracket number
using System;
using System.Collections.Generic;

class GFG
{
    // function to print
```



```
// the bracket number
static void printBracketNumber(string exp,
                                int n)
{
    // used to print the bracket
    // number for the left bracket
    int left_bnum = 1;

    // used to obtain the bracket
    // number for the right bracket
    Stack<int> right_bnum = new Stack<int>();

    // traverse the given
    // expression 'exp'
    for (int i = 0; i < n; i++)
    {

        // if current character
        // is a left bracket
        if (exp[i] == '(')
        {

            // print 'left_bnum',
            Console.Write(left_bnum + " ");

            // Push 'left_bum' on to
            // the stack 'right_bnum'
            right_bnum.Push(left_bnum);

            // increment 'left_bnum' by 1
            left_bnum++;
        }

        // else if current character
        // is a right bracket
        else if(exp[i] == ')')
        {

            // print the top element
            // of stack 'right_bnum'
            // it will be the right
            // bracket number
            Console.Write(right_bnum.Peek() + " ");

            // Pop the top element
            // from the stack
            right_bnum.Pop();
        }
    }
}
```

```
    }  
}  
  
// Driver Code  
static void Main()  
{  
    string exp = "(a+(b*c))+(d/e)";  
    int n = exp.Length;  
  
    printBracketNumber(exp, n);  
}  
  
// This code is contributed  
// by Manish Shaw(manishshaw1)
```

PHP

```
<?php  
// PHP implementation to  
// print the bracket number  
  
// function to print  
// the bracket number  
function printBracketNumber($exp, $n)  
{  
    // used to print the  
    // bracket number for  
    // the left bracket  
    $left_bnum = 1;  
  
    // used to obtain the  
    // bracket number for  
    // the right bracket  
    $right_bnum = array();  
    $t = 0;  
  
    // traverse the given  
    // expression 'exp'  
    for ($i = 0; $i < $n; $i++)  
    {  
  
        // if current character  
        // is a left bracket  
        if ($exp[$i] == '(')  
        {  
  
            // print 'left_bnum',
```

```
        echo $left_bnum . " ";

        // push 'left_bum' on to
        // the stack 'right_bnum'
        $right_bnum[$t++] = $left_bnum;

        // increment 'left_bnum' by 1
        $left_bnum++;
    }

    // else if current character
    // is a right bracket
    else if($exp[$i] == ')')
    {

        // print the top element
        // of stack 'right_bnum'
        // it will be the right
        // bracket number
        echo $right_bnum[$t - 1] . " ";

        // pop the top element
        // from the stack
        $right_bnum[$t - 1] = 1;
        $t--;
    }
}

// Driver Code
$exp = "(a+(b*c))+(d/e)";
$n = strlen($exp);

printBracketNumber($exp, $n);

// This code is contributed
// by mits
?>
```

Output:

1 2 2 1 3 3

Time Complexity : $O(n)$.

Auxiliary Space : $O(n)$.

Improved By : [manishshaw1](#), [Mithun Kumar](#)

Source

<https://www.geeksforgeeks.org/print-bracket-number/>

Chapter 87

Print Reverse a linked list using Stack

Print Reverse a linked list using Stack - GeeksforGeeks

Given a linked list, print reverse of it without modifying the list.

Examples:

```
Input : 1 2 3 4 5 6
Output : 6 5 4 3 2 1
```

```
Input : 12 23 34 45 56 67 78
Output : 78 67 56 45 34 23 12
```

Given a [Linked List](#), display the linked list in reverse without using recursion, stack or modifications to given list.

Examples:

```
Input : 1->2->3->4->5->NULL
Output : 5->4->3->2->1->NULL
```

```
Input : 10->5->15->20->24->NULL
Output : 24->20->15->5->10->NULL
```

Below are different solutions that are now allowed here as we cannot use extra space and modify list.

1) [Recursive solution to print reverse a linked list](#). Requires extra space.

- 2) [Reverse linked list](#) and then print. This requires modifications to original list.
- 3) [A \$O\(n^2\)\$ solution to print reverse of linked list](#) that first count nodes and then prints k-th node from end.

In this post, an efficient stack based solution is discussed.

1. First insert all the element in stack
2. Print stack till stack is not empty

Note: Instead of inserting data from each node into the stack, insert the node's address onto the stack. This is because the size of the node's data will be generally more than the size of the node's address. Thus the stack would end up requiring more memory if it directly stored the data elements. Also, we cannot insert the node's data onto the stack if each node contained more than one data member. Hence the simpler and efficient solution would be to simply insert the node's address.

```
// C/C++ program to print reverse of linked list
// using stack.
#include<bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* Given a reference (pointer to pointer) to the head
of a list and an int, push a new node on the front
of the list. */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Counts no. of nodes in linked list */
int getCount(struct Node* head)
{
    int count = 0; // Initialize count
    struct Node* current = head; // Initialize current
    while (current != NULL)
    {
        count++;
        current = current->next;
    }
}
```

```
    }
    return count;
}

/* Takes head pointer of the linked list and index
   as arguments and return data at index*/
int getNth(struct Node* head, int n)
{
    struct Node* curr = head;
    for (int i=0; i<n-1 && curr != NULL; i++)
        curr = curr->next;
    return curr->data;
}

void printReverse(Node *head)
{
    // store Node addresses in stack
    stack<Node *> stk;
    Node* ptr = head;
    while (ptr != NULL)
    {
        stk.push(ptr);
        ptr = ptr->next;
    }

    // print data from stack
    while (!stk.empty())
    {
        cout << stk.top()->data << " ";
        stk.pop(); // pop after print
    }
    cout << "\n";
}

/* Drier program to test count function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Use push() to construct below list
       1->2->3->4->5 */
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);
}
```

```
    printReverse(head);  
  
    return 0;  
}
```

Output:

5 4 3 2 1

Improved By : [Sayan Mahapatra](#)

Source

<https://www.geeksforgeeks.org/print-reverse-linked-list-using-stack/>

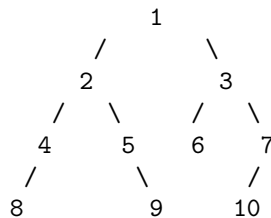
Chapter 88

Print ancestors of a given binary tree node without recursion

Print ancestors of a given binary tree node without recursion - GeeksforGeeks

Given a Binary Tree and a key, write a function that prints all the ancestors of the key in the given binary tree.

For example, consider the following Binary Tree



Following are different input keys and their ancestors in the above tree

Input Key	List of Ancestors
1	
2	1
3	1
4	2 1
5	2 1
6	3 1
7	3 1
8	4 2 1
9	5 2 1
10	7 3 1

Recursive solution for this problem is discussed [here](#).

It is clear that we need to use a stack based iterative traversal of the Binary Tree. The idea is to have all ancestors in stack when we reach the node with given key. Once we reach the key, all we have to do is, print contents of stack.

How to get all ancestors in stack when we reach the given node? We can traverse all nodes in Postorder way. If we take a closer look at the recursive postorder traversal, we can easily observe that, when recursive function is called for a node, the recursion call stack contains ancestors of the node. So idea is do iterative Postorder traversal and stop the traversal when we reach the desired node.

Following is implementation of the above approach.

C

```
// C program to print all ancestors of a given key
#include <stdio.h>
#include <stdlib.h>

// Maximum stack size
#define MAX_SIZE 100

// Structure for a tree node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Structure for Stack
struct Stack
{
    int size;
    int top;
    struct Node* *array;
};

// A utility function to create a new tree node
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// A utility function to create a stack of given size
struct Stack* createStack(int size)
{

```

```
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
    stack->size = size;
    stack->top = -1;
    stack->array = (struct Node**) malloc(stack->size * sizeof(struct Node*));
    return stack;
}

// BASIC OPERATIONS OF STACK
int isFull(struct Stack* stack)
{
    return ((stack->top + 1) == stack->size);
}
int isEmpty(struct Stack* stack)
{
    return stack->top == -1;
}
void push(struct Stack* stack, struct Node* node)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = node;
}
struct Node* pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}
struct Node* peek(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top];
}

// Iterative Function to print all ancestors of a given key
void printAncestors(struct Node *root, int key)
{
    if (root == NULL) return;

    // Create a stack to hold ancestors
    struct Stack* stack = createStack(MAX_SIZE);

    // Traverse the complete tree in postorder way till we find the key
    while (1)
    {
        // Traverse the left side. While traversing, push the nodes into
        // the stack so that their right subtrees can be traversed later
    }
}
```

```
while (root && root->data != key)
{
    push(stack, root);    // push current node
    root = root->left;    // move to next node
}

// If the node whose ancestors are to be printed is found,
// then break the while loop.
if (root && root->data == key)
    break;

// Check if right sub-tree exists for the node at top
// If not then pop that node because we don't need this
// node any more.
if (peek(stack)->right == NULL)
{
    root = pop(stack);

    // If the popped node is right child of top, then remove the top
    // as well. Left child of the top must have processed before.
    // Consider the following tree for example and key = 3. If we
    // remove the following loop, the program will go in an
    // infinite loop after reaching 5.
    //      1
    //     / \
    //    2  3
    //     \
    //      4
    //       \
    //        5
    while (!isEmpty(stack) && peek(stack)->right == root)
        root = pop(stack);
}

// if stack is not empty then simply set the root as right child
// of top and start traversing right sub-tree.
root = isEmpty(stack)? NULL: peek(stack)->right;
}

// If stack is not empty, print contents of stack
// Here assumption is that the key is there in tree
while (!isEmpty(stack))
    printf("%d ", pop(stack)->data);
}

// Driver program to test above functions
int main()
{
```

```
// Let us construct a binary tree
struct Node* root = newNode(1);
root->left = newNode(2);
root->right = newNode(3);
root->left->left = newNode(4);
root->left->right = newNode(5);
root->right->left = newNode(6);
root->right->right = newNode(7);
root->left->left->left = newNode(8);
root->left->right->right = newNode(9);
root->right->right->left = newNode(10);

printf("Following are all keys and their ancestors\n");
for (int key = 1; key <= 10; key++)
{
    printf("%d: ", key);
    printAncestors(root, key);
    printf("\n");
}

getchar();
return 0;
}
```

C++

```
// C++ program to print all ancestors of a given key
#include <bits/stdc++.h>
using namespace std;

// Structure for a tree node
struct Node
{
    int data;
    struct Node *left, *right;
};

// A utility function to create a new tree node
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Iterative Function to print all ancestors of a given key
```

```
void printAncestors(struct Node *root, int key)
{
    if (root == NULL) return;

    // Create a stack to hold ancestors
    stack<struct Node* > st;

    // Traverse the complete tree in postorder way till we find the key
    while (1)
    {
        // Traverse the left side. While traversing, push the nodes into
        // the stack so that their right subtrees can be traversed later
        while (root && root->data != key)
        {
            st.push(root);    // push current node
            root = root->left; // move to next node
        }

        // If the node whose ancestors are to be printed is found,
        // then break the while loop.
        if (root && root->data == key)
            break;

        // Check if right sub-tree exists for the node at top
        // If not then pop that node because we don't need this
        // node any more.
        if (st.top()->right == NULL)
        {
            root = st.top();
            st.pop();

            // If the popped node is right child of top, then remove the top
            // as well. Left child of the top must have processed before.

            while (!st.empty() && st.top()->right == root)
            {
                root = st.top();
                st.pop();
            }
        }

        // if stack is not empty then simply set the root as right child
        // of top and start traversing right sub-tree.
        root = st.empty()? NULL: st.top()->right;
    }

    // If stack is not empty, print contents of stack
    // Here assumption is that the key is there in tree
    while (!st.empty())
```

```
{
    cout<<st.top()->data<<" ";
    st.pop();
}

// Driver program to test above functions
int main()
{
    // Let us construct a binary tree
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->left->left->left = newNode(8);
    root->left->right->right = newNode(9);
    root->right->right->left = newNode(10);

    cout<<"Following are all keys and their ancestors"<<endl;
    for (int key = 1; key <= 10; key++)
    {
        cout<<key<<": "<<" ";
        printAncestors(root, key);
        cout<<endl;
    }

    return 0;
}

// This code is contributed by Gautam Singh
```

Java

```
// Java program to print all ancestors of a given key
import java.util.Stack;

public class GFG
{
    // Class for a tree node
    static class Node
    {
        int data;
        Node left, right;

        // constructor to create Node
        // left and right are by default null
    }
}
```

```
Node(int data)
{
    this.data = data;
}

// Iterative Function to print all ancestors of a given key
static void printAncestors(Node root,int key)
{
    if(root == null)
        return;

    // Create a stack to hold ancestors
    Stack<Node> st = new Stack<>();

    // Traverse the complete tree in postorder way till we find the key
    while(true)
    {

        // Traverse the left side. While traversing, push the nodes into
        // the stack so that their right subtrees can be traversed later
        while(root != null && root.data != key)
        {
            st.push(root);    // push current node
            root = root.left;  // move to next node
        }

        // If the node whose ancestors are to be printed is found,
        // then break the while loop.
        if(root != null && root.data == key)
            break;

        // Check if right sub-tree exists for the node at top
        // If not then pop that node because we don't need this
        // node any more.
        if(st.peek().right == null)
        {
            root =st.peek();
            st.pop();

            // If the popped node is right child of top, then remove the top
            // as well. Left child of the top must have processed before.
            while( st.empty() == false && st.peek().right == root)
            {
                root = st.peek();
                st.pop();
            }
        }
    }
}
```



```
        // if stack is not empty then simply set the root as right child
        // of top and start traversing right sub-tree.
        root = st.empty() ? null : st.peek().right;
    }

    // If stack is not empty, print contents of stack
    // Here assumption is that the key is there in tree
    while( !st.empty() )
    {
        System.out.print(st.peek().data+" ");
        st.pop();
    }
}

// Driver program to test above functions
public static void main(String[] args)
{
    // Let us construct a binary tree
    Node root = new Node(1);
    root.left = new Node(2);
    root.right = new Node(3);
    root.left.left = new Node(4);
    root.left.right = new Node(5);
    root.right.left = new Node(6);
    root.right.right = new Node(7);
    root.left.left.left = new Node(8);
    root.left.right.right = new Node(9);
    root.right.right.left = new Node(10);

    System.out.println("Following are all keys and their ancestors");
    for(int key = 1;key <= 10;key++)
    {
        System.out.print(key+": ");
        printAncestors(root, key);
        System.out.println();
    }
}

//This code is Contributed by Sumit Ghosh
```

Output:

```
Following are all keys and their ancestors
1:
2: 1
3: 1
```

```
4: 2 1
5: 2 1
6: 3 1
7: 3 1
8: 4 2 1
9: 5 2 1
10: 7 3 1
```

Exercise

Note that the above solution assumes that the given key is present in the given Binary Tree. It may go in infinite loop if key is not present. Extend the above solution to work even when the key is not present in tree.

This article is contributed by [Chandra Prakash](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/print-ancestors-of-a-given-binary-tree-node-without-recursion/>

Chapter 89

Print next greater number of Q queries

Print next greater number of Q queries - GeeksforGeeks

Given an array of n elements and q queries, for each query which has an index i, find the next greater element and print its value. If there is no such greater element to its right then print -1.

Examples:

```
Input : arr[] = {3, 4, 2, 7, 5, 8, 10, 6}
        query indexes = {3, 6, 1}
```

```
Output: 8 -1 7
```

Explanation :

For the 1st query index is 3, element is 7 and the next greater element at its right is 8

For the 2nd query index is 6, element is 10 and there is no element greater than 10 at right, so print -1.

For the 3rd query index is 1, element is 4 and the next greater element at its right is 7.

Normal Approach: A normal approach will be for every query move in a loop from index to n and find out the next greater element and print it, but this in worst case will take n iterations, which is a lot if the number of queries are high.

Time Complexity: $O(n^2)$

Auxiliary Space>: $O(1)$

Efficient Approach:

An efficient approach is based on [next greater element](#). We store the index of the next

greater element in an array and for every query process, answer the query in $O(1)$ that will make it more efficient.

But to find out the next greater element for every index in array there are two ways.

One will take $O(n^2)$ and $O(n)$ space which will be to iterate from $I+1$ to n for each element at index I and find out the next greater element and store it.

But the more efficient one will be to use stack, where we use indexes to compare and store in `next[]` the next greater element index.

1) Push the first index to stack.

2) Pick rest of the index one by one and follow following steps in loop.

....a) Mark the current element as i .

....b) If stack is not empty, then pop an index from stack and compare `a[index]` with `a[i]`.

....c) If `a[i]` is greater than the `a[index]`, then `a[i]` is the next greater element for the `a[index]`.

....d) Keep popping from the stack while the popped index element is smaller than `a[i]`. `a[i]` becomes the next greater element for all such popped elements

....g) If `a[i]` is smaller than the popped index element, then push the popped index back.

3) After the loop in step 2 is over, pop all the index from stack and print -1 as next index for them.

C++

```
// C++ program to print
// next greater number
// of Q queries
#include <bits/stdc++.h>
using namespace std;

// array to store the next
// greater element index
void next_greatest(int next[],
                   int a[], int n)
{
    // use of stl
    // stack in c++
    stack<int> s;

    // push the 0th
    // index to the stack
    s.push(0);

    // traverse in the
    // loop from 1-nth index
    for (int i = 1; i < n; i++)
    {

        // iterate till loop is empty
        while (!s.empty()) {
```

```
// get the topmost
// index in the stack
int cur = s.top();

// if the current element is
// greater then the top indexth
// element, then this will be
// the next greatest index
// of the top indexth element
if (a[cur] < a[i])
{
    // initialise the cur
    // index position's
    // next greatest as index
    next[cur] = i;

    // pop the cur index
    // as its greater
    // element has been found
    s.pop();
}

// if not greater
// then break
else
    break;
}
// push the i index so that its
// next greatest can be found
s.push(i);
}

// iterate for all other
// index left inside stack
while (!s.empty())
{
    int cur = s.top();

    // mark it as -1 as no
    // element in greater
    // then it in right
    next[cur] = -1;

    s.pop();
}
}
```

```
// answers all
// queries in O(1)
int answer_query(int a[], int next[],
                 int n, int index)
{
    // stores the next greater
    // element positions
    int position = next[index];

    // if position is -1 then no
    // greater element is at right.
    if (position == -1)
        return -1;

    // if there is a index that
    // has greater element
    // at right then return its
    // value as a[position]
    else
        return a[position];
}

// Driver Code
int main()
{
    int a[] = {3, 4, 2, 7,
               5, 8, 10, 6 };

    int n = sizeof(a) / sizeof(a[0]);

    // initializes the
    // next array as 0
    int next[n] = { 0 };

    // calls the function
    // to pre-calculate
    // the next greatest
    // element indexes
    next_greatest(next, a, n);

    // query 1 answered
    cout << answer_query(a, next, n, 3) << " ";

    // query 2 answered
    cout << answer_query(a, next, n, 6) << " ";

    // query 3 answered
```

```
    cout << answer_query(a, next, n, 1) << " ";
}
```

Java

```
// Java program to print
// next greater number
// of Q queries
import java.util.*;

class GFG
{
    public static int[] query(int arr[],
                              int query[])
    {
        int ans[] = new int[arr.length]; // this array contains
                                           // the next greatest
                                           // elements of all the elements

        Stack<Integer> s = new Stack<>();
        // push the 0th index
        // to the stack
        s.push(arr[0]);
        int j = 0;
        //traverse rest
        // of the array
        for(int i = 1; i < arr.length; i++)
        {
            int next = arr[i];

            if(!s.isEmpty())
            {
                // get the topmost
                // element in the stack
                int element = s.pop();

                /* If the popped element
                is smaller than next,
                then a) store the pair
                b) keep popping while
                elements are smaller and
                stack is not empty */
                while(next > element)
                {
                    ans[j] = next;
                    j++;
                    if(s.isEmpty())
                        break;
                    element = s.pop();
                }
            }
        }
    }
}
```

```
    }

    /* If element is greater
    than next, then
    push the element back */
    if (element > next)
        s.push(element);
    }
    /* push next to stack so
    that we can find next
    greater for it */
    s.push(next);
}
/* After iterating over the
loop, the remaining elements
in stack do not have the next
greater element, so -1 for them */
while(!s.isEmpty())
{
    int element = s.pop();
    ans[j] = -1;
    j++;
}

// return the next
// greatest array
return ans;
}

// Driver Code
public static void main(String[] args)
{
    int arr[] = {3, 4, 2, 7,
                 5, 8, 10, 6};
    int query[] = {3, 6, 1};
    int ans[] = query(arr, query);

    // getting output array
    // with next greatest elements
    for(int i = 0; i < query.length; i++)
    {
        // displaying the next greater
        // element for given set of queries
        System.out.print(ans[query[i]] + " ");
    }
}
}
```



```
// This code was contributed  
// by Harshit Sood
```

Output:

8 -1 7

Time complexity: $\max(O(n), O(q))$, $O(n)$ for pre-processing the next[] array and $O(1)$ for every query.

Auxiliary Space: $O(n)$

Improved By : [HarshitSood1](#)

Source

<https://www.geeksforgeeks.org/print-next-greater-number-q-queries/>

Chapter 90

Program for Tower of Hanoi

Program for Tower of Hanoi - GeeksforGeeks

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

- 1) Only one disk can be moved at a time.
- 2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- 3) No disk may be placed on top of a smaller disk.

Approach :

Take an example for 2 disks :

Let rod 1 = 'A', rod 2 = 'B', rod 3 = 'C'.

Step 1 : Shift first disk from 'A' to 'B'.

Step 2 : Shift second disk from 'A' to 'C'.

Step 3 : Shift first disk from 'B' to 'C'.

The pattern here is :

Shift 'n-1' disks from 'A' to 'B'.

Shift last disk from 'A' to 'C'.

Shift 'n-1' disks from 'B' to 'C'.

Image illustration for 3 disks :

Examples:

Input : 2

Output : Disk 1 moved from A to B

```
Disk 2 moved from A to C
Disk 1 moved from B to C
```

Input : 3

```
Output : Disk 1 moved from A to C
         Disk 2 moved from A to B
         Disk 1 moved from C to B
         Disk 3 moved from A to C
         Disk 1 moved from B to A
         Disk 2 moved from B to C
         Disk 1 moved from A to C
```

C/C++

```
#include <stdio.h>

// C recursive function to solve tower of hanoi puzzle
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
    if (n == 1)
    {
        printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
        return;
    }
    towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
    printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
    towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main()
{
    int n = 4; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
    return 0;
}
```

Java

```
// Java recursive program to solve tower of hanoi puzzle

class GFG
{
    // Java recursive function to solve tower of hanoi puzzle
    static void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
    {
        if (n == 1)
        {
```

```
        System.out.println("Move disk 1 from rod " + from_rod + " to rod " + to_rod);
        return;
    }
    towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
    System.out.println("Move disk " + n + " from rod " + from_rod + " to rod " + to_rod);
    towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

// Driver method
public static void main(String args[])
{
    int n = 4; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
}
}
```

Python

```
# Recursive Python function to solve tower of hanoi

def TowerOfHanoi(n , from_rod, to_rod, aux_rod):
    if n == 1:
        print "Move disk 1 from rod",from_rod,"to rod",to_rod
        return
    TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)
    print "Move disk",n,"from rod",from_rod,"to rod",to_rod
    TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)

# Driver code
n = 4
TowerOfHanoi(n, 'A', 'C', 'B')
# A, C, B are the name of rods

# Contributed By Harshit Agrawal
```

PHP

```
<?php
//PHP code to solve Tower of Hanoi problem.

// Recursive Function to solve Tower of Hanoi
function towerOfHanoi($n, $from_rod, $to_rod, $aux_rod) {

    if ($n === 1) {
        echo ("Move disk 1 from rod $from_rod to rod $to_rod \n");
        return;
    }
}
```

```
towerOfHanoi($n-1, $from_rod, $aux_rod, $to_rod);
echo ("Move disk $n from rod $from_rod to rod $to_rod \n");
towerOfHanoi($n-1, $aux_rod, $to_rod, $from_rod);

}

// Driver code

// number of disks
$n = 4;

// A, B and C are names of rods
towerOfHanoi($n, 'A', 'C', 'B');

// This code is contributed by akash7981
?>

C#

// C# recursive program to solve
// tower of hanoi puzzle
using System;

class geek
{
    // C# recursive function to solve
    // tower of hanoi puzzle
    static void towerOfHanoi(int n, char from_rod,
                             char to_rod, char aux_rod)
    {
        if (n == 1)
        {
            Console.WriteLine("Move disk 1 from rod " + from_rod
                              + " to rod " + to_rod);

            return;
        }
        towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
        Console.WriteLine("Move disk " + n + " from rod "
                          + from_rod + " to rod " + to_rod);
        towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
    }

    // Driver method
    public static void Main()
    {
        // Number of disks
```

```
int n = 4;

// A, B and C are names of rods
towerOfHanoi(n, 'A', 'C', 'B');
}

// This code is contributed by Sam007
```

Output:

```
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 2 from rod B to rod A
Move disk 1 from rod C to rod A
Move disk 3 from rod B to rod C
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
```

For n disks, total $2^n - 1$ moves are required.

eg: For 4 disks $2^4 - 1 = 15$ moves are required.

For n disks, total $2^{n+1} - 1$ function calls are made.

eg: For 4 disks $2^{4+1} - 1 = 31$ function calls are made.

Related Articles

- [Recursive Functions](#)
- [Iterative solution to TOH puzzle](#)
- [Quiz on Recursion](#)

References:

http://en.wikipedia.org/wiki/Tower_of_Hanoi

Improved By : [vaibhav29498](#), [Rustam Ali](#)

Source

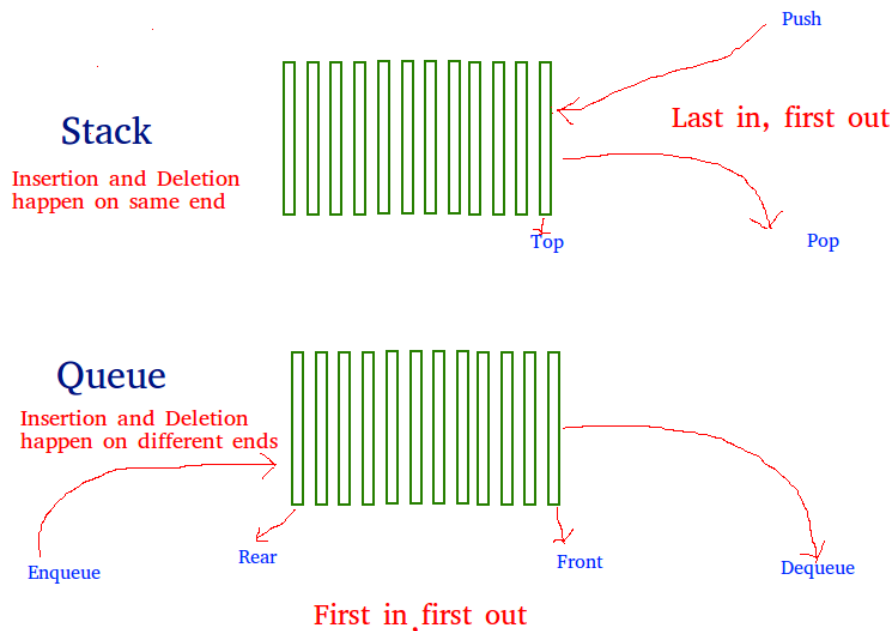
<https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/>

Chapter 91

Queue using Stacks

Queue using Stacks - GeeksforGeeks

The problem is opposite of [this](#) post. We are given a stack data structure with push and pop operations, the task is to implement a queue using instances of stack data structure and operations on them.



A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be $stack1$ and $stack2$. q can be implemented in two ways:

Method 1 (By making enqueue operation costly) This method makes sure that oldest entered element is always at the top of $stack1$, so that dequeue operation just pops from $stack1$. To put the element at top of $stack1$, $stack2$ is used.

enqueue(q, x)

- 1) While stack1 is not empty, push everything from stack1 to stack2.
- 2) Push x to stack1 (assuming size of stacks is unlimited).
- 3) Push everything back to stack1.

Here time complexity will be $O(n)$

dequeue(q)

- 1) If stack1 is empty then error
- 2) Pop an item from stack1 and return it

Here time complexity will be $O(1)$

C++

```
// CPP program to implement Queue using
// two stacks with costly enqueue()
#include <bits/stdc++.h>
using namespace std;

struct Queue {
    stack<int> s1, s2;

    void enqueue(int x)
    {
        // Move all elements from s1 to s2
        while (!s1.empty()) {
            s2.push(s1.top());
            s1.pop();
        }

        // Push item into s1
        s1.push(x);

        // Push everything back to s1
        while (!s2.empty()) {
            s1.push(s2.top());
            s2.pop();
        }
    }

    // Dequeue an item from the queue
    int dequeue()
    {
        // if first stack is empty
        if (s1.empty()) {
            cout << "Q is Empty";
            exit(0);
        }
    }
}
```



```
        // Return top of s1
        int x = s1.top();
        s1.pop();
        return x;
    }
};

// Driver code
int main()
{
    Queue q;
    q.enqueue(1);
    q.enqueue(2);
    q.enqueue(3);

    cout << q.dequeue() << '\n';
    cout << q.dequeue() << '\n';
    cout << q.dequeue() << '\n';

    return 0;
}
```

Method 2 (By making dequeue operation costly) In this method, in enqueue operation, the new element is entered at the top of stack1. In dequeue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

enqueue(q, x)

1) Push x to stack1 (assuming size of stacks is unlimited).

Here time complexity will be $O(1)$

dequeue(q)

1) If both stacks are empty then error.

2) If stack2 is empty

While stack1 is not empty, push everything from stack1 to stack2.

3) Pop the element from stack2 and return it.

Here time complexity will be $O(n)$

Method 2 is definitely better than method 1.

Method 1 moves all the elements twice in enqueue operation, while method 2 (in dequeue operation) moves the elements once and moves elements only if stack2 is empty.

Implementation of method 2:

C++

```
// CPP program to implement Queue using
// two stacks with costly dequeue()
```

```
#include <bits/stdc++.h>
using namespace std;

struct Queue {
    stack<int> s1, s2;

    // Enqueue an item to the queue
    void enqueue(int x)
    {
        // Push item into the first stack
        s1.push(x);
    }

    // Dequeue an item from the queue
    int dequeue()
    {
        // if both stacks are empty
        if (s1.empty() && s2.empty()) {
            cout << "Q is empty";
            exit(0);
        }

        // if s2 is empty, move
        // elements from s1
        if (s2.empty()) {
            while (!s1.empty()) {
                s2.push(s1.top());
                s1.pop();
            }
        }

        // return the top item from s2
        int x = s2.top();
        s2.pop();
        return x;
    }
};

// Driver code
int main()
{
    Queue q;
    q.enqueue(1);
    q.enqueue(2);
    q.enqueue(3);

    cout << q.dequeue() << '\n';
    cout << q.dequeue() << '\n';
}
```

```
    cout << q.deQueue() << '\n';

    return 0;
}
```

C

```
/* C Program to implement a queue using two stacks */
#include <stdio.h>
#include <stdlib.h>

/* structure of a stack node */
struct sNode {
    int data;
    struct sNode* next;
};

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data);

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref);

/* structure of queue having two stacks */
struct queue {
    struct sNode* stack1;
    struct sNode* stack2;
};

/* Function to enqueue an item to queue */
void enQueue(struct queue* q, int x)
{
    push(&q->stack1, x);
}

/* Function to deQueue an item from queue */
int deQueue(struct queue* q)
{
    int x;

    /* If both stacks are empty then error */
    if (q->stack1 == NULL && q->stack2 == NULL) {
        printf("Q is empty");
        getchar();
        exit(0);
    }

    /* Move elements from stack1 to stack 2 only if
```

```
        stack2 is empty */
    if (q->stack2 == NULL) {
        while (q->stack1 != NULL) {
            x = pop(&q->stack1);
            push(&q->stack2, x);
        }
    }

    x = pop(&q->stack2);
    return x;
}

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data)
{
    /* allocate node */
    struct sNode* new_node = (struct sNode*)malloc(sizeof(struct sNode));
    if (new_node == NULL) {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*top_ref);

    /* move the head to point to the new node */
    (*top_ref) = new_node;
}

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref)
{
    int res;
    struct sNode* top;

    /*If stack is empty then error */
    if (*top_ref == NULL) {
        printf("Stack underflow \n");
        getchar();
        exit(0);
    }
    else {
        top = *top_ref;
        res = top->data;
    }
}
```

```
        *top_ref = top->next;
        free(top);
        return res;
    }
}

/* Driver function to test anove functions */
int main()
{
    /* Create a queue with items 1 2 3*/
    struct queue* q = (struct queue*)malloc(sizeof(struct queue));
    q->stack1 = NULL;
    q->stack2 = NULL;
    enqueue(q, 1);
    enqueue(q, 2);
    enqueue(q, 3);

    /* Dequeue items */
    printf("%d ", dequeue(q));
    printf("%d ", dequeue(q));
    printf("%d ", dequeue(q));

    return 0;
}
```

Java

```
/* Java Program to implement a queue using two stacks */
// Note that Stack class is used for Stack implementation

import java.util.Stack;

public class GFG {
    /* class of queue having two stacks */
    static class Queue {
        Stack<Integer> stack1;
        Stack<Integer> stack2;
    }

    /* Function to push an item to stack*/
    static void push(Stack<Integer> top_ref, int new_data)
    {
        // Push the data onto the stack
        top_ref.push(new_data);
    }

    /* Function to pop an item from stack*/
    static int pop(Stack<Integer> top_ref)
```

```
{
    /*If stack is empty then error */
    if (top_ref.isEmpty()) {
        System.out.println("Stack Underflow");
        System.exit(0);
    }

    // pop the data from the stack
    return top_ref.pop();
}

// Function to enqueue an item to the queue
static void enqueue(Queue q, int x)
{
    push(q.stack1, x);
}

/* Function to dequeue an item from queue */
static int dequeue(Queue q)
{
    int x;

    /* If both stacks are empty then error */
    if (q.stack1.isEmpty() && q.stack2.isEmpty()) {
        System.out.println("Q is empty");
        System.exit(0);
    }

    /* Move elements from stack1 to stack 2 only if
    stack2 is empty */
    if (q.stack2.isEmpty()) {
        while (!q.stack1.isEmpty()) {
            x = pop(q.stack1);
            push(q.stack2, x);
        }
    }
    x = pop(q.stack2);
    return x;
}

/* Driver function to test above functions */
public static void main(String args[])
{
    /* Create a queue with items 1 2 3*/
    Queue q = new Queue();
    q.stack1 = new Stack<>();
    q.stack2 = new Stack<>();
    enqueue(q, 1);
```

```
        enqueue(q, 2);
        enqueue(q, 3);

        /* Dequeue items */
        System.out.print(deQueue(q) + " ");
        System.out.print(deQueue(q) + " ");
        System.out.println(deQueue(q) + " ");
    }
}
// This code is contributed by Sumit Ghosh
```

Output:

1 2 3

Queue can also be implemented using one user stack and one Function Call Stack. Below is modified Method 2 where recursion (or Function Call Stack) is used to implement queue using only one user defined stack.

```
enqueue(x)
    1) Push x to stack1.

deQueue:
    1) If stack1 is empty then error.
    2) If stack1 has only one element then return it.
    3) Recursively pop everything from the stack1, store the popped item
        in a variable res, push the res back to stack1 and return res
```

The step 3 makes sure that the last popped item is always returned and since the recursion stops when there is only one item in *stack1* (step 2), we get the last element of *stack1* in *deQueue()* and all other items are pushed back in step

3. Implementation of method 2 using Function Call Stack:

C++

```
// CPP program to implement Queue using
// one stack and recursive call stack.
#include <bits/stdc++.h>
using namespace std;

struct Queue {
    stack<int> s;

    // Enqueue an item to the queue
    void enqueue(int x)
    {
```

```
        s.push(x);
    }

    // Dequeue an item from the queue
    int dequeue()
    {
        if (s.empty()) {
            cout << "Q is empty";
            exit(0);
        }

        // pop an item from the stack
        int x = s.top();
        s.pop();

        // if stack becomes empty, return
        // the popped item
        if (s.empty())
            return x;

        // recursive call
        int item = dequeue();

        // push popped item back to the stack
        s.push(x);

        // return the result of dequeue() call
        return item;
    }
};

// Driver code
int main()
{
    Queue q;
    q.enqueue(1);
    q.enqueue(2);
    q.enqueue(3);

    cout << q.dequeue() << '\n';
    cout << q.dequeue() << '\n';
    cout << q.dequeue() << '\n';

    return 0;
}
```

C


```
/* Program to implement a queue using one user defined stack
and one Function Call Stack */
#include <stdio.h>
#include <stdlib.h>

/* structure of a stack node */
struct sNode {
    int data;
    struct sNode* next;
};

/* structure of queue having two stacks */
struct queue {
    struct sNode* stack1;
};

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data);

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref);

/* Function to enqueue an item to queue */
void enQueue(struct queue* q, int x)
{
    push(&q->stack1, x);
}

/* Function to dequeue an item from queue */
int dequeue(struct queue* q)
{
    int x, res;

    /* If both stacks are empty then error */
    if (q->stack1 == NULL) {
        printf("Q is empty");
        getchar();
        exit(0);
    }
    else if (q->stack1->next == NULL) {
        return pop(&q->stack1);
    }
    else {
        /* pop an item from the stack1 */
        x = pop(&q->stack1);

        /* store the last dequeued item */
        res = dequeue(q);
    }
}
```

```
        /* push everything back to stack1 */
        push(&q->stack1, x);
        return res;
    }
}

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data)
{
    /* allocate node */
    struct sNode* new_node = (struct sNode*)malloc(sizeof(struct sNode));

    if (new_node == NULL) {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*top_ref);

    /* move the head to point to the new node */
    (*top_ref) = new_node;
}

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref)
{
    int res;
    struct sNode* top;

    /*If stack is empty then error */
    if (*top_ref == NULL) {
        printf("Stack underflow \n");
        getchar();
        exit(0);
    }
    else {
        top = *top_ref;
        res = top->data;
        *top_ref = top->next;
        free(top);
        return res;
    }
}
```

```
}

/* Driver function to test above functions */
int main()
{
    /* Create a queue with items 1 2 3*/
    struct queue* q = (struct queue*)malloc(sizeof(struct queue));
    q->stack1 = NULL;

    enqueue(q, 1);
    enqueue(q, 2);
    enqueue(q, 3);

    /* Dequeue items */
    printf("%d ", dequeue(q));
    printf("%d ", dequeue(q));
    printf("%d ", dequeue(q));

    return 0;
}
```

Java

```
// Java Program to implement a queue using one stack

import java.util.Stack;

public class QOneStack {
    // class of queue having two stacks
    static class Queue {
        Stack<Integer> stack1;
    }

    /* Function to push an item to stack*/
    static void push(Stack<Integer> top_ref, int new_data)
    {
        /* put in the data */
        top_ref.push(new_data);
    }

    /* Function to pop an item from stack*/
    static int pop(Stack<Integer> top_ref)
    {
        /*If stack is empty then error */
        if (top_ref == null) {
            System.out.println("Stack Underflow");
            System.exit(0);
        }
    }
}
```

```
        // return element from stack
        return top_ref.pop();
    }

    /* Function to enqueue an item to queue */
    static void enqueue(Queue q, int x)
    {
        push(q.stack1, x);
    }

    /* Function to dequeue an item from queue */
    static int dequeue(Queue q)
    {
        int x, res = 0;
        /* If the stacks is empty then error */
        if (q.stack1.isEmpty()) {
            System.out.println("Q is Empty");
            System.exit(0);
        }
        // Check if it is a last element of stack
        else if (q.stack1.size() == 1) {
            return pop(q.stack1);
        }
        else {

            /* pop an item from the stack1 */
            x = pop(q.stack1);

            /* store the last dequeued item */
            res = dequeue(q);

            /* push everything back to stack1 */
            push(q.stack1, x);
            return res;
        }
        return 0;
    }

    /* Driver function to test above functions */
    public static void main(String[] args)
    {
        /* Create a queue with items 1 2 3*/
        Queue q = new Queue();
        q.stack1 = new Stack<>();

        enqueue(q, 1);
        enqueue(q, 2);
        enqueue(q, 3);
    }
}
```

```
        /* Dequeue items */
        System.out.print(deQueue(q) + " ");
        System.out.print(deQueue(q) + " ");
        System.out.print(deQueue(q) + " ");
    }
}
// This code is contributed by Sumit Ghosh
```

Output:

1 2 3

Please write comments if you find any of the above codes/algorithms incorrect, or find better ways to solve the same problem.

Improved By : [iamsdhar](#), [ParulShandilya](#)

Source

<https://www.geeksforgeeks.org/queue-using-stacks/>

Chapter 92

Range Queries for Longest Correct Bracket Subsequence

Range Queries for Longest Correct Bracket Subsequence - GeeksforGeeks

Given a bracket sequence or in other words a string S of length n, consisting of characters '(' and ')'. Find length of the maximum correct bracket subsequence of sequence for a given query range. *Note: A correct bracket sequence is the one that have matched bracket pairs or which contains another nested correct bracket sequence. For e.g (), (()), ()() are some correct bracket sequence.*

Examples:

```
Input : S = ())(())(())(
        Start Index of Range = 0,
        End Index of Range = 11
Output : 10
Explanation: Longest Correct Bracket Subsequence is ()(())(())
```

```
Input : S = ())(())(())(
        Start Index of Range = 1,
        End Index of Range = 2
Output : 0
```

[Segment Trees](#) can be used to solve this problem **efficiently**

At each node of the segment tree, we store the following:

- 1) a - Number of correctly matched pairs of brackets.
- 2) b - Number of unused open brackets.
- 3) c - Number of unused closed brackets.

(unused open bracket – means they can't be matched with any closing bracket, unused closed bracket – means they can't be matched with any opening bracket, for e.g $S =) ($ contains an unused open and an unused closed bracket)

For each interval $[L, R]$, we can match X number of unused open brackets '(' in interval $[L, MID]$ with unused closed brackets ')' in interval $[MID + 1, R]$ where

$X = \text{minimum}(\text{number of unused '(' in } [L, MID], \text{ number of unused ')' in } [MID + 1, R])$

Hence, X is also the number of correctly matched pairs built by combination.

So, for interval $[L, R]$

1) Total number of correctly matched pairs becomes the sum of correctly matched pairs in left child and correctly matched pairs in right child and number of combinations of unused '(' and unused ')' from left and right child respectively.

$$a[L, R] = a[L, MID] + a[MID + 1, R] + X$$

2) Total number of unused open brackets becomes the sum of unused open brackets in left child and unused open brackets in right child minus X (minus – because we used X unused '(' from left child to match with unused ')' from right child).

$$a[L, R] = b[L, MID] + b[MID + 1, R] - X$$

3) Similarly, for unused closed brackets, following relation holds.

$$a[L, R] = c[L, MID] + c[MID + 1, R] - X$$

where a , b and c are the representations described above for each node to be stored in.

Below is the implementation of above approach in C++.

```
/* CPP Program to find the longest correct
   bracket subsequence in a given range */
#include <bits/stdc++.h>
using namespace std;

/* Declaring Structure for storing
   three values in each segment tree node */
struct Node {
    int pairs;
    int open; // unused
    int closed; // unused

    Node()
    {
        pairs = open = closed = 0;
    }
};
```

```
// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) { return s + (e - s) / 2; }

// Returns Parent Node after merging its left and right child
Node merge(Node leftChild, Node rightChild)
{
    Node parentNode;
    int minMatched = min(leftChild.open, rightChild.closed);
    parentNode.pairs = leftChild.pairs + rightChild.pairs + minMatched;
    parentNode.open = leftChild.open + rightChild.open - minMatched;
    parentNode.closed = leftChild.closed + rightChild.closed - minMatched;
    return parentNode;
}

// A recursive function that constructs Segment Tree
// for string[ss..se]. si is index of current node in
// segment tree st
void constructSTUtil(char str[], int ss, int se, Node* st,
                    int si)
{
    // If there is one element in string, store it in
    // current node of segment tree and return
    if (ss == se) {

        // since it contains one element, pairs
        // will be zero
        st[si].pairs = 0;

        // check whether that one element is opening
        // bracket or not
        st[si].open = (str[ss] == '(' ? 1 : 0);

        // check whether that one element is closing
        // bracket or not
        st[si].closed = (str[ss] == ')' ? 1 : 0);

        return;
    }

    // If there are more than one elements, then recur
    // for left and right subtrees and store the relation
    // of values in this node
    int mid = getMid(ss, se);
    constructSTUtil(str, ss, mid, st, si * 2 + 1);
    constructSTUtil(str, mid + 1, se, st, si * 2 + 2);

    // Merge left and right child into the Parent Node
    st[si] = merge(st[si * 2 + 1], st[si * 2 + 2]);
}
```



```
}

/* Function to construct segment tree from given
   string. This function allocates memory for segment
   tree and calls constructSTUtil() to fill the
   allocated memory */
Node* constructST(char str[], int n)
{
    // Allocate memory for segment tree

    // Height of segment tree
    int x = (int)(ceil(log2(n)));

    // Maximum size of segment tree
    int max_size = 2 * (int)pow(2, x) - 1;

    // Declaring array of structure Allocate memory
    Node* st = new Node[max_size];

    // Fill the allocated memory st
    constructSTUtil(str, 0, n - 1, st, 0);

    // Return the constructed segment tree
    return st;
}

/* A Recursive function to get the desired
   Maximum Sum Sub-Array,
   The following are parameters of the function-

   st    --> Pointer to segment tree
   si --> Index of the segment tree Node
   ss & se --> Starting and ending indexes of the
               segment represented by
               current Node, i.e., tree[index]
   qs & qe --> Starting and ending indexes of query range */
Node queryUtil(Node* st, int ss, int se, int qs,
               int qe, int si)
{
    // No overlap
    if (ss > qe || se < qs) {

        // returns a Node for out of bounds condition
        Node nullNode;
        return nullNode;
    }

    // Complete overlap
```

```
    if (ss >= qs && se <= qe) {
        return st[si];
    }

    // Partial Overlap Merge results of Left
    // and Right subtrees
    int mid = getMid(ss, se);
    Node left = queryUtil(st, ss, mid, qs, qe, si * 2 + 1);
    Node right = queryUtil(st, mid + 1, se, qs, qe, si * 2 + 2);

    // merge left and right subtree query results
    Node res = merge(left, right);
    return res;
}

/* Returns the maximum length correct bracket
subsequence between start and end
It mainly uses queryUtil(). */
int query(Node* st, int qs, int qe, int n)
{
    Node res = queryUtil(st, 0, n - 1, qs, qe, 0);

    // since we are storing numbers pairs
    // and have to return maximum length, hence
    // multiply no of pairs by 2
    return 2 * res.pairs;
}

// Driver Code
int main()
{
    char str[] = "()()()()()(";
    int n = strlen(str);

    // Build segment tree from given string
    Node* st = constructST(str, n);

    int startIndex = 0, endIndex = 11;
    cout << "Maximum Length Correct Bracket"
         << " Subsequence between "
         << startIndex << " and " << endIndex << " = "
         << query(st, startIndex, endIndex, n) << endl;

    startIndex = 1, endIndex = 2;
    cout << "Maximum Length Correct Bracket"
         << " Subsequence between "
         << startIndex << " and " << endIndex << " = "
         << query(st, startIndex, endIndex, n) << endl;
```

```
    return 0;  
}
```

Output:

Maximum Length Correct Bracket Subsequence between 0 and 11 = 10

Maximum Length Correct Bracket Subsequence between 1 and 2 = 0

Time complexity for each query is $O(\log N)$, where N is the size of string.

Source

<https://www.geeksforgeeks.org/range-queries-longest-correct-bracket-subsequence/>

Chapter 93

Range Queries for Longest Correct Bracket Subsequence Set | 2

Range Queries for Longest Correct Bracket Subsequence Set | 2 - GeeksforGeeks

Given a bracket sequence or in other words a string S of length n, consisting of characters '(' and ')'. Find length of the maximum correct bracket subsequence of sequence for a given query range. Note: A correct bracket sequence is the one that have matched bracket pairs or which contains another nested correct bracket sequence. For e.g (), (()), ()() are some correct bracket sequence.

Examples:

```
Input : S = ()()()()()(  
        Start Index of Range = 0,  
        End Index of Range = 11  
Output : 10  
Explanation: Longest Correct Bracket Subsequence is ()()()()
```

```
Input : S = ()()()()()(  
        Start Index of Range = 1,  
        End Index of Range = 2  
Output : 0
```

Approach : In the Previous post ([SET 1](#)) we discussed a solution that works in $O(\text{long})$ for each query, now in this post we will go to see a solution that works in $O(1)$ for each query.

Idea is based on the Post [length of the longest valid balanced substring](#) If we marked indexes of all Balanced parentheses/bracket in a temporary array (here we named it BCP[], BOP[]

) then we answer each query in $O(1)$ time.

Algorithm :

```
stack is used to get the index of balance Bracket
Traverse a string from 0 ..to n
IF we seen a closing bracket,
    ( i.e., str[i] = ')' && stack is not empty )
```

```
Then mark both "open & close" bracket indexes as 1 .
BCP[i] = 1;
BOP[stk.top()] = 1;
```

```
And At least stored cumulative sum of BCP[] & BOP[]
Run a loop from 1 to n
BOP[i] +=BOP[i-1], BCP[i] +=BCP[i-1]
```

Now you can answer each query in $O(1)$ time

```
(BCP[e] - (BOP[s-1] - BCP[s-1] ) ) * 2;
```

Below is the implementation of above idea.

```
// CPP code to answer the query in constant time
#include<bits/stdc++.h>
using namespace std;

/*
BOP[] stands for "Balanced open parentheses"
BCP[] stands for "Balanced close parentheses"
*/

// function for precomputation
void constructBalanceArray( int BOP[] , int BCP[] ,
                           char* str , int n )
{
    // Create a stack and push -1 as initial index to it.
    stack<int> stk;

    // Initialize result
    int result = 0;

    // Traverse all characters of given string
    for (int i=0; i<n; i++)
```

```

{
    // If opening bracket, push index of it
    if (str[i] == '(')
        stk.push(i);

    else // If closing bracket, i.e., str[i] = ')'
    {
        // If closing bracket, i.e., str[i] = ')'
        // && stack is not empty then mark both
        // "open & close" bracket indexes as 1 .
        // Pop the previous opening bracket's index
        if (!stk.empty())
        {
            BCP[i] = 1;
            BOP[stk.top()] = 1;
            stk.pop();
        }

        // If stack is empty.
        else
            BCP[i] = 0;
    }
}

for( int i = 1 ; i < n; i++ )
{
    BCP[i] += BCP[i-1];
    BOP[i] += BOP[i-1];
}

}

// Function return output of each query in O(1)
int query( int BOP[] , int BCP[] ,
           int s , int e )
{
    if( s != 0 )
        return (BCP[e] - (BOP[s-1] - BCP[s-1] ) ) * 2;
    else
        return BCP[e] * 2;
}

// Driver program to test above function
int main()
{
    char str[] = "())(())(())(";
    int n = strlen(str);

```

```
int BCP[n+1] ={0};
int BOP[n+1] ={0};

constructBlanceArray( BOP , BCP , str, n );

int startIndex = 5, endIndex = 11;

cout << "Maximum Length Correct Bracket"
      << " Subsequence between "
      << startIndex << " and " << endIndex << " = "
      << query( BOP , BCP, startIndex, endIndex ) << endl;

startIndex = 4, endIndex = 5;
cout << "Maximum Length Correct Bracket"
      << " Subsequence between "
      << startIndex << " and " << endIndex << " = "
      << query( BOP ,BCP, startIndex, endIndex ) << endl;

      startIndex = 1, endIndex = 5;
cout << "Maximum Length Correct Bracket"
      << " Subsequence between "
      << startIndex << " and " << endIndex << " = "
      << query( BOP ,BCP, startIndex, endIndex ) << endl;

return 0;
}
```

Output:

```
Maximum Length Correct Bracket Subsequence between 5 and 11 = 6
Maximum Length Correct Bracket Subsequence between 4 and 5 = 2
Maximum Length Correct Bracket Subsequence between 1 and 5 = 2
```

Time complexity for each query is $O(1)$.

Source

<https://www.geeksforgeeks.org/range-queries-longest-correct-bracket-subsequence-set-2/>

Chapter 94

Rat in a Maze | Backtracking using Stack

Rat in a Maze | Backtracking using Stack - GeeksforGeeks

Prerequisites – [Recursion](#), [Backtracking](#) and [Stack Data Structure](#).

A Maze is given as N*M binary matrix of blocks and there is a rat initially at (0, 0) ie. maze[0][0] and the rat wants to eat food which is present at some given block in the maze (fx, fy). In a maze matrix, 0 means that the block is a dead end and 1 means that the block can be used in the path from source to destination. The rat can move in any direction (not diagonally) to any block provided the block is not a dead end.

The task is to check if there exists any path so that the rat can reach the food or not. It is not needed to print the path.

Examples:

```
Input : maze[4][5] = {
    {1, 0, 1, 1, 0},
    {1, 1, 1, 0, 1},
    {0, 1, 0, 1, 1},
    {1, 1, 1, 1, 1}
}
```

```
fx = 2, fy=3
```

Output : Path Found!

The path can be: (0, 0) -> (1, 0) -> (1, 1) -> (2, 1) -> (3, 1) -> (3, 2) -> (3, 3) -> (2, 3)

This is the famous [Rat in a Maze](#) problem asked in many interviews that can be solved using Recursion and Backtracking. We already have discussed a Backtracking solution to this problem using recursion in [Rat in a Maze | Backtracking-2](#). In this an iterative solution using stack is discussed.

In the [previous article](#), [Recursion](#) uses a call stack to keep the store each recursive call and then pop as the function ends. We will eliminate recursion by using our own stack to do the same thing.

A node structure is used to store the (i, j) coordinates and directions explored from this node and which direction to try out next.

Structure Used:

1. X : x coordinate of the node
2. Y : y coordinate of the node
3. dir : This variable will be used to tell which all directions we have tried and which to choose next. We will try all the directions in anti-clockwise manner starting from Up. Initially it will be assigned 0.
 - If dir=0 try Up direction.
 - If dir=1 try left direction.
 - If dir=2 try down direction.
 - If dir=3 try right direction.

Initially, we will push a node with indexes i=0, j=0 and dir=0 into the stack. We will move to all the direction of the topmost node one by one in an anti-clockwise manner and each time as we try out a new path we will push that node (block of the maze) in the stack. We will increase *dir* variable of the topmost node each time so that we can try a new direction each time unless all the directions are explored ie. dir=4. If dir equals to 4 we will pop that node from the stack that means we are retracting one step back to the path where we came from.

We will also maintain a visited matrix which will maintain which blocks of the maze are already used in the path or in other words present in the stack. While trying out any direction we will also check if the block of the maze is not a dead end and is not out of the maze too.

We will do this while either the topmost node coordinates become equal to the food's coordinates that means we have reached the food or the stack becomes empty which means that there is no possible path to reach the food.

Below is the implementation of the above approach:

```
// CPP program to solve Rat in a maze
// problem with backtracking using stack

#include <cstring>
#include <iostream>
#include <stack>

using namespace std;

#define N 4
#define M 5
```

```
class node {
public:
    int x, y;
    int dir;

    node(int i, int j)
    {
        x = i;
        y = j;

        // Initially direction
        // set to 0
        dir = 0;
    }
};

// maze of n*m matrix
int n = N, m = M;

// Coordinates of food
int fx, fy;
bool visited[N][M];

bool isReachable(int maze[N][M])
{
    // Initially starting at (0, 0).
    int i = 0, j = 0;

    stack<node> s;

    node temp(i, j);

    s.push(temp);

    while (!s.empty()) {

        // Pop the top node and move to the
        // left, right, top, down or retract
        // back according the value of node's
        // dir variable.
        temp = s.top();
        int d = temp.dir;
        i = temp.x, j = temp.y;

        // Increment the direction and
        // push the node in the stack again.
        temp.dir++;
    }
}
```

```
s.pop();
s.push(temp);

// If we reach the Food coordinates
// return true
if (i == fx and j == fy) {
    return true;
}

// Checking the Up direction.
if (d == 0) {
    if (i - 1 >= 0 and maze[i - 1][j] and
        visited[i - 1][j]) {
        node temp1(i - 1, j);
        visited[i - 1][j] = false;
        s.push(temp1);
    }
}

// Checking the left direction
else if (d == 1) {
    if (j - 1 >= 0 and maze[i][j - 1] and
        visited[i][j - 1]) {
        node temp1(i, j - 1);
        visited[i][j - 1] = false;
        s.push(temp1);
    }
}

// Checking the down direction
else if (d == 2) {
    if (i + 1 < n and maze[i + 1][j] and
        visited[i + 1][j]) {
        node temp1(i + 1, j);
        visited[i + 1][j] = false;
        s.push(temp1);
    }
}

// Checking the right direction
else if (d == 3) {
    if (j + 1 < m and maze[i][j + 1] and
        visited[i][j + 1]) {
        node temp1(i, j + 1);
        visited[i][j + 1] = false;
        s.push(temp1);
    }
}
```

```
        // If none of the direction can take
        // the rat to the Food, retract back
        // to the path where the rat came from.
        else {
            visited[temp.x][temp.y] = true;
            s.pop();
        }
    }

    // If the stack is empty and
    // no path is found return false.
    return false;
}

int main()
{
    // Initially setting the visited
    // array to true (unvisited)
    memset(visited, true, sizeof(visited));

    // Maze matrix
    int maze[N][M] = {
        { 1, 0, 1, 1, 0 },
        { 1, 1, 1, 0, 1 },
        { 0, 1, 0, 1, 1 },
        { 1, 1, 1, 1, 1 }
    };

    // Food coordinates
    fx = 2;
    fy = 3;

    if (isReachable(maze)) {
        cout << "Path Found!" << '\n';
    }
    else
        cout << "No Path Found!" << '\n';

    return 0;
}
```

Output:

Path Found!

Note: We can also print the path by just popping the nodes out of the stacks and then

print them in reverse order.

Improved By : [dhruvgupta167](#)

Source

<https://www.geeksforgeeks.org/rat-in-a-maze-backtracking-using-stack/>

Chapter 95

Remove brackets from an algebraic string containing + and – operators

Remove brackets from an algebraic string containing + and - operators - GeeksforGeeks

Simplify a given algebraic string of characters, '+', '-' operators and parentheses. Output the simplified string without parentheses.

Examples:

Input : "a-(b+c)"

Output : "a-b-c"

Input : "a-(b-c-(d+e))-f"

Output : "a-b+c+d+e-f"

The idea is to check operators just before starting of bracket, i.e., before character '('. If operator is -, we need to toggle all operators inside the bracket. A stack is used which stores only two integers 0 and 1 to indicate whether to toggle or not.

We iterate for every character of input string. Initially push 0 to stack. Whenever the character is an operator ('+' or '-'), check top of stack. If top of stack is 0, append the same operator in the resultant string. If top of stack is 1, append the other operator (if '+' append '-' in the resultant string).

```
// C++ program to simplify algebraic string
#include <bits/stdc++.h>
using namespace std;
```

```
// Function to simplify the string
```

```
char* simplify(string str)
{
    int len = str.length();

    // resultant string of max length equal
    // to length of input string
    char* res = new char(len);
    int index = 0, i = 0;

    // create empty stack
    stack<int> s;
    s.push(0);

    while (i < len) {
        if (str[i] == '+') {

            // If top is 1, flip the operator
            if (s.top() == 1)
                res[index++] = '-';

            // If top is 0, append the same operator
            if (s.top() == 0)
                res[index++] = '+';

        } else if (str[i] == '-') {
            if (s.top() == 1)
                res[index++] = '+';
            else if (s.top() == 0)
                res[index++] = '-';
        } else if (str[i] == '(' && i > 0) {
            if (str[i - 1] == '-') {

                // x is opposite to the top of stack
                int x = (s.top() == 1) ? 0 : 1;
                s.push(x);
            }

            // push value equal to top of the stack
            else if (str[i - 1] == '+')
                s.push(s.top());
        }

        // If closing parentheses pop the stack once
        else if (str[i] == ')')
            s.pop();

        // copy the character to the result
        else
```

```
        res[index++] = str[i];
        i++;
    }
    return res;
}

// Driver program
int main()
{
    string s1 = "a-(b+c)";
    string s2 = "a-(b-c-(d+e))-f";
    cout << simplify(s1) << endl;
    cout << simplify(s2) << endl;
    return 0;
}
```

Output:

```
a-b-c
a-b+c+d+e-f
```

Source

<https://www.geeksforgeeks.org/remove-brackets-algebraic-string-containing-operators/>

Chapter 96

Remove repeated digits in a given number

Remove repeated digits in a given number - GeeksforGeeks

Given an integer, remove consecutive repeated digits from it.

Examples:

Input: x = 12224
Output: 124

Input: x = 124422
Output: 1242

Input: x = 11332
Output: 132

We need to process all digits of n and remove consecutive representations. We can go through all digits by repeatedly dividing n with 10 and taking $n\%10$.

C++

```
// C++ program to remove repeated digits
#include <iostream>
using namespace std;

long int removeRecur(long int n)
{
    // Store first digits as previous digit
    int prev_digit = n % 10;
```

```
// Initialize power
long int pow = 10;
long int res = prev_digit;

// Iterate through all digits of n, note that
// the digits are processed from least significant
// digit to most significant digit.
while (n) {
    // Store current digit
    int curr_digit = n % 10;

    if (curr_digit != prev_digit) {
        // Add the current digit to the beginning
        // of result
        res += curr_digit * pow;

        // Update previous result and power
        prev_digit = curr_digit;
        pow *= 10;
    }

    // Remove last digit from n
    n = n / 10;
}

return res;
}

// Drive program
int main()
{
    long int n = 12224;
    cout << removeRecur(n);
    return 0;
}
```

Java

```
// Java program to remove repeated digits
import java.io.*;

class GFG {

    static long removeRecur(long n)
    {

        // Store first digits as previous
```

```
// digit
long prev_digit = n % 10;

// Initialize power
long pow = 10;
long res = prev_digit;

// Iterate through all digits of n,
// note that the digits are
// processed from least significant
// digit to most significant digit.
while (n>0) {

    // Store current digit
    long curr_digit = n % 10;

    if (curr_digit != prev_digit)
    {
        // Add the current digit to
        // the beginning of result
        res += curr_digit * pow;

        // Update previous result
        // and power
        prev_digit = curr_digit;
        pow *= 10;
    }

    // Remove last digit from n
    n = n / 10;
}

return res;
}

// Drive program
public static void main (String[] args)
{
    long n = 12224;

    System.out.println(removeRecur(n));
}

// This code is contributed by anuj_67.
```

C#

```
// C# program to remove repeated digits
using System;

class GFG {

    static long removeRecur(long n)
    {

        // Store first digits as previous
        // digit
        long prev_digit = n % 10;

        // Initialize power
        long pow = 10;
        long res = prev_digit;

        // Iterate through all digits of n,
        // note that the digits are
        // processed from least significant
        // digit to most significant digit.
        while (n > 0) {

            // Store current digit
            long curr_digit = n % 10;

            if (curr_digit != prev_digit)
            {
                // Add the current digit to
                // the beginning of result
                res += curr_digit * pow;

                // Update previous result
                // and power
                prev_digit = curr_digit;
                pow *= 10;
            }

            // Remove last digit from n
            n = n / 10;
        }

        return res;
    }

    // Drive program
    public static void Main ()
    {
        long n = 12224;
```

```
        Console.WriteLine(removeRecur(n));
    }
}

// This code is contributed by anuj_67.
```

PHP

```
<?php
// PHP program to remove
// repeated digits

function removeRecur($n)
{
    // Store first digits
    // as previous digit
    $prev_digit = $n % 10;

    // Initialize power
    $pow = 10;
    $res = $prev_digit;

    // Iterate through all digits
    // of n, note that the digits
    // are processed from least
    // significant digit to most
    // significant digit.
    while ($n)
    {
        // Store current digit
        $curr_digit = $n%10;

        if ($curr_digit != $prev_digit)
        {
            // Add the current digit
            // to the beginning of
            // result
            $res += $curr_digit * $pow;

            // Update previous result
            // and power
            $prev_digit = $curr_digit;
            $pow *= 10;
        }
    }
}
```

```
        // Remove last digit
        // from n
        $n = $n / 10;
    }

    return $res;
}

// Driver Code
$n = 12224;
echo removeRecur($n);

// This ocde is contributed by ajit.
?>
```

Output:

124

This article is contributed by Kartik. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : [jit_t](#), [vt_m](#)

Source

<https://www.geeksforgeeks.org/remove-repeated-digits-in-a-given-number/>

Chapter 97

Reverse a number using stack

Reverse a number using stack - GeeksforGeeks

Given a number , write a program to reverse this number using stack.

Examples:

Input : 365
Output : 563

Input : 6899
Output : 9986

We have already discussed the simple method to reverse a number in [this](#) post. In this post we will discuss about how to reverse a number using stack.

The idea to do this is to extract digits of the number and push the digits on to a stack. Once all of the digits of the number are pushed to the stack, we will start popping the contents of stack one by one and form a number.

As stack is a LIFO data structure, digits of the newly formed number will be in reverse order.

Below is the C++ implementation of above idea:

C++

```
// CPP program to reverse the number
// using a stack

#include <bits/stdc++.h>
using namespace std;

// Stack to maintain order of digits
```

```
stack <int> st;

// Function to push digits into stack
void push_digits(int number)
{
    while (number != 0)
    {
        st.push(number % 10);
        number = number / 10;
    }
}

// Function to reverse the number
int reverse_number(int number)
{
    // Function call to push number's
    // digits to stack
    push_digits(number);

    int reverse = 0;
    int i = 1;

    // Popping the digits and forming
    // the reversed number
    while (!st.empty())
    {
        reverse = reverse + (st.top() * i);
        st.pop();
        i = i * 10;
    }

    // Return the reversed number formed
    return reverse;
}

// Driver program to test above function
int main()
{
    int number = 39997;

    // Function call to reverse number
    cout << reverse_number(number);

    return 0;
}
```

Java


```
// Java program to reverse the number
// using a stack
import java.util.Stack;

public class GFG
{
    // Stack to maintain order of digits
    static Stack<Integer> st= new Stack<>();

    // Function to push digits into stack
    static void push_digits(int number)
    {
        while(number != 0)
        {
            st.push(number % 10);
            number = number / 10;
        }
    }

    // Function to reverse the number
    static int reverse_number(int number)
    {
        // Function call to push number's
        // digits to stack
        push_digits(number);
        int reverse = 0;
        int i = 1;

        // Popping the digits and forming
        // the reversed number
        while (!st.isEmpty())
        {
            reverse = reverse + (st.peek() * i);
            st.pop();
            i = i * 10;
        }

        // Return the reversed number formed
        return reverse;
    }

    // Driver program to test above function
    public static void main(String[] args)
    {
        int number = 39997;
        System.out.println(reverse_number(number));
    }
}
```

```
// This code is contributed by Sumit Ghosh
```

Output:

79993

Time Complexity: $O(\log N)$

Auxiliary Space: $O(\log N)$, Where N is the input number.

Source

<https://www.geeksforgeeks.org/reverse-number-using-stack/>

Chapter 98

Reverse a stack using recursion

Reverse a stack using recursion - GeeksforGeeks

Write a program to reverse a stack using recursion. You are not allowed to use loop constructs like while, for..etc, and you can only use the following ADT functions on Stack S:

isEmpty(S)

push(S)

pop(S)

The idea of the solution is to hold all values in Function Call Stack until the stack becomes empty. When the stack becomes empty, insert all held items one by one at the bottom of the stack.

For example, let the input stack be

```
1 <-- top
2
3
4
```

First 4 is inserted at the bottom.

```
4 <-- top
```

Then 3 is inserted at the bottom

```
4 <-- top
3
```

Then 2 is inserted at the bottom

```
4 <-- top
3
2
```

Then 1 is inserted at the bottom

```
4 <-- top
3
2
1
```

So we need a function that inserts at the bottom of a stack using the above given basic stack function.

void insertAtBottom(): First pops all stack items and stores the popped item in function call stack using recursion. And when stack becomes empty, pushes new item and all items stored in call stack.

void reverse(): This function mainly uses insertAtBottom() to pop all items one by one and insert the popped items at the bottom.

C

```
// C program to reverse a
// stack using recursion
#include<stdio.h>
#include<stdlib.h>
#define bool int

// structure of a stack node
struct sNode
{
    char data;
    struct sNode *next;
};

// Function Prototypes
void push(struct sNode** top_ref,
          int new_data);
int pop(struct sNode** top_ref);
bool isEmpty(struct sNode* top);
void print(struct sNode* top);

// Below is a recursive function
// that inserts an element
// at the bottom of a stack.
void insertAtBottom(struct sNode** top_ref,
                    int item)
{
    if (isEmpty(*top_ref))
        push(top_ref, item);
    else
    {
```

```
        // Hold all items in Function Call
        // Stack until we reach end of the
        // stack. When the stack becomes
        // empty, the isEmpty(*top_ref) becomes
        // true, the above if part is executed
        // and the item is inserted at the bottom
        int temp = pop(top_ref);
        insertAtBottom(top_ref, item);

        // Once the item is inserted
        // at the bottom, push all
        // the items held in Function
        // Call Stack
        push(top_ref, temp);
    }
}
```

```
// Below is the function that
// reverses the given stack using
// insertAtBottom()
void reverse(struct sNode** top_ref)
{
    if (!isEmpty(*top_ref))
    {
        // Hold all items in Function
        // Call Stack until we
        // reach end of the stack
        int temp = pop(top_ref);
        reverse(top_ref);

        // Insert all the items (held in
        // Function Call Stack)
        // one by one from the bottom
        // to top. Every item is
        // inserted at the bottom
        insertAtBottom(top_ref, temp);
    }
}
```

```
// Driver Code
int main()
{
    struct sNode *s = NULL;
    push(&s, 4);
    push(&s, 3);
    push(&s, 2);
    push(&s, 1);
}
```

```
    printf("\n Original Stack ");
    print(s);
    reverse(&s);
    printf("\n Reversed Stack ");
    print(s);
    return 0;
}

// Function to check if
// the stack is empty
bool isEmpty(struct sNode* top)
{
    return (top == NULL)? 1 : 0;
}

// Function to push an item to stack
void push(struct sNode** top_ref,
          int new_data)
{
    // allocate node
    struct sNode* new_node =
        (struct sNode*) malloc(sizeof(struct sNode));

    if (new_node == NULL)
    {
        printf("Stack overflow \n");
        exit(0);
    }

    // put in the data
    new_node->data = new_data;

    // link the old list
    // off the new node
    new_node->next = (*top_ref);

    // move the head to
    // point to the new node
    (*top_ref) = new_node;
}

// Function to pop an item from stack
int pop(struct sNode** top_ref)
{
    char res;
    struct sNode *top;
```

```
// If stack is empty then error
if (*top_ref == NULL)
{
    printf("Stack overflow \n");
    exit(0);
}
else
{
    top = *top_ref;
    res = top->data;
    *top_ref = top->next;
    free(top);
    return res;
}
}
```

```
// Function to print a
// linked list
void print(struct sNode* top)
{
    printf("\n");
    while (top != NULL)
    {
        printf(" %d ", top->data);
        top = top->next;
    }
}
```

C++

```
// C++ code to reverse a
// stack using recursion
#include<bits/stdc++.h>
using namespace std;

// using std::stack for
// stack implementation
stack<char> st;

// intializing a string to store
// result of reversed stack
string ns;

// Below is a recursive function
// that inserts an element
// at the bottom of a stack.
char insert_at_bottom(char x)
{

```

```
if(st.size() == 0)
    st.push(x);

else
{
    // All items are held in Function Call
    // Stack until we reach end of the stack
    // When the stack becomes empty, the
    // st.size() becomes 0, the above if
    // part is executed and the item is
    // inserted at the bottom

    char a = st.top();
    st.pop();
    insert_at_bottom(x);

    // push all the items held in
    // Function Call Stack
    // once the item is inserted
    // at the bottom
    st.push(a);
}
}

// Below is the function that
// reverses the given stack using
// insert_at_bottom()
char reverse()
{
    if(st.size() > 0)
    {
        // Hold all items in Function
        // Call Stack until we
        // reach end of the stack
        char x = st.top();
        st.pop();
        reverse();

        // Insert all the items held
        // in Function Call Stack
        // one by one from the bottom
        // to top. Every item is
        // inserted at the bottom
        insert_at_bottom(x);
    }
}
```



```
}

// Driver Code
int main()
{
    // push elements into
    // the stack
    st.push('1');
    st.push('2');
    st.push('3');
    st.push('4');

    cout<<"Original Stack"<<endl;

    // print the elements
    // of original stack
    cout<<"1"<<" "<<"2"<<" "
        <<"3"<<" "<<"4"
        <<endl;

    // function to reverse
    // the stack
    reverse();
    cout<<"Reversed Stack"
        <<endl;

    // storing values of reversed
    // stack into a string for display
    while(!st.empty())
    {
        char p=st.top();
        st.pop();
        ns+=p;
    }

    //display of reversed stack
    cout<<ns[3]<<" "<<ns[2]<<" "
        <<ns[1]<<" "<<ns[0]<<endl;
    return 0;
}

// This code is contributed by Gautam Singh
```

Java

```
// Java code to reverse a
// stack using recursion
```

```
import java.util.Stack;

class Test {

    // using Stack class for
    // stack implementation
    static Stack<Character> st = new Stack<>();

    // Below is a recursive function
    // that inserts an element
    // at the bottom of a stack.
    static void insert_at_bottom(char x)
    {

        if(st.isEmpty())
            st.push(x);

        else
        {

            // All items are held in Function
            // Call Stack until we reach end
            // of the stack. When the stack becomes
            // empty, the st.size() becomes 0, the
            // above if part is executed and
            // the item is inserted at the bottom
            char a = st.peek();
            st.pop();
            insert_at_bottom(x);

            // push all the items held
            // in Function Call Stack
            // once the item is inserted
            // at the bottom
            st.push(a);
        }
    }

    // Below is the function that
    // reverses the given stack using
    // insert_at_bottom()
    static void reverse()
    {
        if(st.size() > 0)
        {

            // Hold all items in Function
            // Call Stack until we
```

```
        // reach end of the stack
        char x = st.peek();
        st.pop();
        reverse();

        // Insert all the items held
        // in Function Call Stack
        // one by one from the bottom
        // to top. Every item is
        // inserted at the bottom
        insert_at_bottom(x);
    }
}

// Driver Code
public static void main(String[] args)
{

    // push elements into
    // the stack
    st.push('1');
    st.push('2');
    st.push('3');
    st.push('4');

    System.out.println("Original Stack");

    System.out.println(st);

    // function to reverse
    // the stack
    reverse();

    System.out.println("Reversed Stack");

    System.out.println(st);
}
}
```

Python3

```
# Python program to reverse a
# stack using recursion

# Below is a recursive function
# that inserts an element
# at the bottom of a stack.
def insertAtBottom(stack, item):
```

```
    if isEmpty(stack):
        push(stack, item)
    else:
        temp = pop(stack)
        insertAtBottom(stack, item)
        push(stack, temp)

# Below is the function that
# reverses the given stack
# using insertAtBottom()
def reverse(stack):
    if not isEmpty(stack):
        temp = pop(stack)
        reverse(stack)
        insertAtBottom(stack, temp)

# Below is a complete running
# program for testing above
# functions.

# Function to create a stack.
# It initializes size of stack
# as 0
def createStack():
    stack = []
    return stack

# Function to check if
# the stack is empty
def isEmpty( stack ):
    return len(stack) == 0

# Function to push an
# item to stack
def push( stack, item ):
    stack.append( item )

# Function to pop an
# item from stack
def pop( stack ):

    # If stack is empty
    # then error
    if(isEmpty( stack )):
        print("Stack Underflow ")
        exit(1)

    return stack.pop()
```

```
# Function to print the stack
def prints(stack):
    for i in range(len(stack)-1, -1, -1):
        print(stack[i], end = ' ')
    print()
```

Driver Code

```
stack = createStack()
push( stack, str(4) )
push( stack, str(3) )
push( stack, str(2) )
push( stack, str(1) )
print("Original Stack ")
prints(stack)
```

```
reverse(stack)
```

```
print("Reversed Stack ")
prints(stack)
```

This code is contributed by Sunny Karira

Output:

```
Original Stack
1 2 3 4
Reversed Stack
4 3 2 1
```

Improved By : [SBanzal](#)

Source

<https://www.geeksforgeeks.org/reverse-a-stack-using-recursion/>

Chapter 99

Reverse a stack without using extra space in $O(n)$

Reverse a stack without using extra space in $O(n)$ - GeeksforGeeks

Reverse a [Stack](#) without using recursion and extra space .Even the functional Stack is not allowed.

Examples:

Input : 1->2->3->4
Output : 4->3->2->1

Input : 6->5->4
Output : 4->5->6

We have discussed a way of reversing a string in below post.

[Reverse a Stack using Recursion](#)

The above solution requires $O(n)$ extra space. We can reverse a string in $O(1)$ time if we internally represent the stack as linked list. Reverse a stack would require reversing a linked list which can be done with $O(n)$ time and $O(1)$ extra space.

Note that push() and pop() operations still take $O(1)$ time.

C++

```
// CPP program to implement Stack
// using linked list so that reverse
// can be done with  $O(1)$  extra space.
#include<bits/stdc++.h>
```

```
using namespace std;

class StackNode {
public:
    int data;
    StackNode *next;

    StackNode(int data)
    {
        this->data = data;
        this->next = NULL;
    }
};

class Stack {

    StackNode *top;

public:

    // Push and pop operations
    void push(int data)
    {
        if (top == NULL) {
            top = new StackNode(data);
            return;
        }
        StackNode *s = new StackNode(data);
        s->next = top;
        top = s;
    }

    StackNode* pop()
    {
        StackNode *s = top;
        top = top->next;
        return s;
    }

    // prints contents of stack
    void display()
    {
        StackNode *s = top;
        while (s != NULL) {
            cout << s->data << " ";
            s = s->next;
        }
        cout << endl;
    }
};
```

```
    }

    // Reverses the stack using simple
    // linked list reversal logic.
    void reverse()
    {
        StackNode *prev, *cur, *succ;
        cur = prev = top;
        cur = cur->next;
        prev->next = NULL;
        while (cur != NULL) {

            succ = cur->next;
            cur->next = prev;
            prev = cur;
            cur = succ;
        }
        top = prev;
    }
};

// driver code
int main()
{
    Stack *s = new Stack();
    s->push(1);
    s->push(2);
    s->push(3);
    s->push(4);
    cout << "Original Stack" << endl;;
    s->display();
    cout << endl;

    // reverse
    s->reverse();

    cout << "Reversed Stack" << endl;
    s->display();

    return 0;
}
// This code is contribute by Chhavi.
```

Java

```
// Java program to implement Stack using linked
// list so that reverse can be done with  $O(1)$ 
// extra space.
```



```
class StackNode {
    int data;
    StackNode next;
    public StackNode(int data)
    {
        this.data = data;
        this.next = null;
    }
}

class Stack {
    StackNode top;

    // Push and pop operations
    public void push(int data)
    {
        if (this.top == null) {
            top = new StackNode(data);
            return;
        }
        StackNode s = new StackNode(data);
        s.next = this.top;
        this.top = s;
    }
    public StackNode pop()
    {
        StackNode s = this.top;
        this.top = this.top.next;
        return s;
    }

    // prints contents of stack
    public void display()
    {
        StackNode s = this.top;
        while (s != null) {
            System.out.print(" " + s.data);
            s = s.next;
        }
        System.out.println();
    }

    // Reverses the stack using simple
    // linked list reversal logic.
    public void reverse()
    {
        StackNode prev, cur, succ;
        cur = prev = this.top;
```

```
        cur = cur.next;
        prev.next = null;
        while (cur != null) {

            succ = cur.next;
            cur.next = prev;
            prev = cur;
            cur = succ;
        }
        this.top = prev;
    }
}

public class reverseStackWithoutSpace {
    public static void main(String[] args)
    {
        Stack s = new Stack();
        s.push(1);
        s.push(2);
        s.push(3);
        s.push(4);
        System.out.println("Original Stack");
        s.display();

        // reverse
        s.reverse();

        System.out.println("Reversed Stack");
        s.display();
    }
}
```

Output:

```
Original Stack
4 3 2 1
Reversed Stack
1 2 3 4
```

Source

<https://www.geeksforgeeks.org/reverse-stack-without-using-extra-space/>

Chapter 100

Reverse individual words

Reverse individual words - GeeksforGeeks

Given a string str, we need to print reverse of individual words.

Examples:

Input : Hello World
Output : olleH dlroW

Input : Geeks for Geeks
Output : skeeG rof skeeG

Method 1 (Simple): Generate all words separated by space. One by one reverse words and print them separated by space.

Method 2 (Space Efficient): We use a stack to push all words before space. As soon as we encounter a space, we empty the stack.

C++

```
// C++ program to reverse individual words in a given
// string using STL list
#include <bits/stdc++.h>
using namespace std;

// reverses individual words of a string
void reverseWords(string str)
{
    stack<char> st;

    // Traverse given string and push all characters
```

```
// to stack until we see a space.
for (int i = 0; i < str.length(); ++i) {
    if (str[i] != ' ')
        st.push(str[i]);

    // When we see a space, we print contents
    // of stack.
    else {
        while (st.empty() == false) {
            cout << st.top();
            st.pop();
        }
        cout << " ";
    }
}

// Since there may not be space after
// last word.
while (st.empty() == false) {
    cout << st.top();
    st.pop();
}
}
```

```
// Driver program to test function
int main()
{
    string str = "Geeks for Geeks";
    reverseWords(str);
    return 0;
}
```

Java

```
// Java program to reverse individual
// words in a given string using STL list
import java.io.*;
import java.util.*;

class GFG {

    // reverses individual words of a string
    static void reverseWords(String str)
    {
        Stack<Character> st=new Stack<Character>();

        // Traverse given string and push all
        // characters to stack until we see a space.
    }
}
```

```
for (int i = 0; i < str.length(); ++i) {
    if (str.charAt(i) != ' ')
        st.push(str.charAt(i));

    // When we see a space, we print
    // contents of stack.
    else {
        while (st.empty() == false) {
            System.out.print(st.pop());

            }
        System.out.print(" ");
    }
}

// Since there may not be space after
// last word.
while (st.empty() == false) {
    System.out.print(st.pop());
}
}

// Driver program to test above function
public static void main(String[] args)
{
    String str = "Geeks for Geeks";
    reverseWords(str);
}
}
```

Output:

skeeG rof skeeG

[Python | Reverse each word in a sentence](#)

Using [stringstream](#) in C++ :

```
#include<bits/stdc++.h>
using namespace std;

void printWords(string str)
{
    // word variable to store word
    string word;

    // making a string stream
```

```
stringstream iss(str);

// Read and print each word.
while (iss >> word){
    reverse(word.begin(),word.end());
    cout<<word<<" ";
}

// Driver code
int main()
{
    string s = "GeeksforGeeks is good to learn";
    printWords(s);
    return 0;
}
// This code is contributed by Nikhil Rawat
```

Time complexity : $O(n)$
Space complexity : $O(n)$

Source

<https://www.geeksforgeeks.org/reverse-individual-words/>

Chapter 101

Reversing a Queue

Reversing a Queue - GeeksforGeeks

Give an algorithm for reversing a queue Q. Only following standard operations are allowed on queue.

1. enqueue(x) : Add an item x to rear of queue.
2. dequeue() : Remove an item from front of queue.
3. empty() : Checks if a queue is empty or not.

Examples:

Input : Q = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Output :Q = [100, 90, 80, 70, 60, 50, 40, 30, 20, 10]

Input :[1, 2, 3, 4, 5]
Output :[5, 4, 3, 2, 1]

C++

```
// CPP program to reverse a Queue
#include <bits/stdc++.h>
using namespace std;

// Utility function to print the queue
void Print(queue<int>& Queue)
{
    while (!Queue.empty()) {
        cout << Queue.front() << " ";
        Queue.pop();
    }
}
```

```
}

// Function to reverse the queue
void reverseQueue(queue<int>& Queue)
{
    stack<int> Stack;
    while (!Queue.empty()) {
        Stack.push(Queue.front());
        Queue.pop();
    }
    while (!Stack.empty()) {
        Queue.push(Stack.top());
        Stack.pop();
    }
}

// Driver code
int main()
{
    queue<int> Queue;
    Queue.push(10);
    Queue.push(20);
    Queue.push(30);
    Queue.push(40);
    Queue.push(50);
    Queue.push(60);
    Queue.push(70);
    Queue.push(80);
    Queue.push(90);
    Queue.push(100);

    reverseQueue(Queue);
    Print(Queue);
}
```

Java

```
// Java program to reverse a Queue
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

// Java program to reverse a queue
public class Queue_reverse {

    static Queue<Integer> queue;

    // Utility function to print the queue
```



```
static void Print()
{
    while (!queue.isEmpty()) {
        System.out.print( queue.peek() + ", ");
        queue.remove();
    }
}

// Function to reverse the queue
static void reversequeue()
{
    Stack<Integer> stack = new Stack<>();
    while (!queue.isEmpty()) {
        stack.add(queue.peek());
        queue.remove();
    }
    while (!stack.isEmpty()) {
        queue.add(stack.peek());
        stack.pop();
    }
}

// Driver code
public static void main(String args[])
{
    queue = new LinkedList<Integer>();
    queue.add(10);
    queue.add(20);
    queue.add(30);
    queue.add(40);
    queue.add(50);
    queue.add(60);
    queue.add(70);
    queue.add(80);
    queue.add(90);
    queue.add(100);

    reversequeue();
    Print();
}
//This code is contributed by Sumit Ghosh
```

Output

100, 90, 80, 70, 60, 50, 40, 30, 20, 10

Source

<https://www.geeksforgeeks.org/reversing-a-queue/>

Chapter 102

Reversing the first K elements of a Queue

Reversing the first K elements of a Queue - GeeksforGeeks

Given an integer k and a [queue](#) of integers, we need to reverse the order of the first k elements of the queue, leaving the other elements in the same relative order.

Only following standard operations are allowed on queue.

- enqueue(x) : Add an item x to rear of queue
- dequeue() : Remove an item from front of queue
- size() : Returns number of elements in queue.
- front() : Finds front item.

Examples:

Input : Q = [10, 20, 30, 40, 50, 60,
70, 80, 90, 100]

k = 5

Output : Q = [50, 40, 30, 20, 10, 60,
70, 80, 90, 100]

Input : Q = [10, 20, 30, 40, 50, 60,
70, 80, 90, 100]

k = 4

Output : Q = [40, 30, 20, 10, 50, 60,
70, 80, 90, 100]

The idea is to use an auxiliary [stack](#).

- 1) Create an empty stack.
- 2) One by one dequeue items from given queue and push the dequeued items to stack.

- 3) Enqueue the contents of stack at the back of the queue
- 4) Reverse the whole queue.

C++

```
// C++ program to reverse first k elements of a queue.
#include <bits/stdc++.h>
using namespace std;

/* Function to reverse the first K elements of the Queue */
void reverseQueueFirstKElements(int k, queue<int>& Queue)
{
    if (Queue.empty() == true || k > Queue.size())
        return;
    if (k <= 0)
        return;

    stack<int> Stack;

    /* Push the first K elements into a Stack*/
    for (int i = 0; i < k; i++) {
        Stack.push(Queue.front());
        Queue.pop();
    }

    /* Enqueue the contents of stack
       at the back of the queue*/
    while (!Stack.empty()) {
        Queue.push(Stack.top());
        Stack.pop();
    }

    /* Remove the remaining elements and
       enqueue them at the end of the Queue*/
    for (int i = 0; i < Queue.size() - k; i++) {
        Queue.push(Queue.front());
        Queue.pop();
    }
}

/* Utility Function to print the Queue */
void Print(queue<int>& Queue)
{
    while (!Queue.empty()) {
        cout << Queue.front() << " ";
        Queue.pop();
    }
}
```

```
// Driver code
int main()
{
    queue<int> Queue;
    Queue.push(10);
    Queue.push(20);
    Queue.push(30);
    Queue.push(40);
    Queue.push(50);
    Queue.push(60);
    Queue.push(70);
    Queue.push(80);
    Queue.push(90);
    Queue.push(100);

    int k = 5;
    reverseQueueFirstKElements(k, Queue);
    Print(Queue);
}
```

Java

```
// Java program to reverse first k elements
// of a queue.
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class Reverse_k_element_queue {

    static Queue<Integer> queue;

    // Function to reverse the first K elements
    // of the Queue
    static void reverseQueueFirstKElements(int k) {
        if (queue.isEmpty() == true || k > queue.size())
            return;
        if (k <= 0)
            return;

        Stack<Integer> stack = new Stack<Integer>();

        // Push the first K elements into a Stack
        for (int i = 0; i < k; i++) {
            stack.push(queue.peek());
            queue.remove();
        }
    }
}
```

```
// Enqueue the contents of stack at the back
// of the queue
while (!stack.empty()) {
    queue.add(stack.peek());
    stack.pop();
}

// Remove the remaining elements and enqueue
// them at the end of the Queue
for (int i = 0; i < queue.size() - k; i++) {
    queue.add(queue.peek());
    queue.remove();
}

// Utility Function to print the Queue
static void Print() {
    while (!queue.isEmpty()) {
        System.out.print(queue.peek() + " ");
        queue.remove();
    }
}

// Driver code
public static void main(String args[]) {
    queue = new LinkedList<Integer>();
    queue.add(10);
    queue.add(20);
    queue.add(30);
    queue.add(40);
    queue.add(50);
    queue.add(60);
    queue.add(70);
    queue.add(80);
    queue.add(90);
    queue.add(100);

    int k = 5;
    reverseQueueFirstKElements(k);
    Print();
}

// This code is contributed by Sumit Ghosh
```

Output:

50 40 30 20 10 60 70 80 90 100

Source

<https://www.geeksforgeeks.org/reversing-first-k-elements-queue/>

Chapter 103

Simplify the directory path (Unix like)

Simplify the directory path (Unix like) - GeeksforGeeks

Given an absolute path for a file (Unix-style), simplify it. Note that absolute path always begin with '/' (root directory), a dot in path represent current directory and double dot represents parent directory.

Examples:

```
"/a/./"    --> means stay at the current directory 'a'
"/a/b/.."  --> means jump to the parent directory
              from 'b' to 'a'
"/////"    --> consecutive multiple '/' are a valid
              path, they are equivalent to single "/".
```

```
Input  : /home/
Output : /home
```

```
Input  : /a/./b/../../../../c/
Output : /c
```

```
Input  : /a/..
Output : /
```

```
Input  : /a/../
Output : /
```

```
Input  : /../../../../../../a
Output : /a
```



```
Input : /a/./b/./c/./d/
Output : /a/b/c/d
```

```
Input : /a/../../../../.
Output : /
```

```
Input : /a//b//c/////d
Output : /a/b/c/d
```

By looking at examples we can see that the above simplification process just behaves like a **stack**. Whenever we encounter any file's name, we simply push it into the stack. when we come across "." we do nothing. When we find "." in our path, we simply pop the topmost element as we have to jump back to parent's directory.

When we see multiple "///" we just ignore them as they are equivalent to one single "/". After iterating through the whole string the elements remaining in the stack is our simplified absolute path. We have to create another stack to reverse the elements stored inside the original stack and then store the result inside a string.

```
/* C++ program to simplify a Unix
   styled absolute path of a file */
#include <bits/stdc++.h>
using namespace std;

// function to simplify a Unix - styled
// absolute path
string simplify(string A)
{
    // stack to store the file's names.
    stack<string> st;

    // temporary string which stores the extracted
    // directory name or commands "." / ".."
    // Eg. "/a/b/./."
    // dir will contain "a", "b", "..", ".";
    string dir;

    // contains resultant simplifies string.
    string res;

    // every string starts from root directory.
    res.append("/");

    // stores length of input string.
    int len_A = A.length();

    for (int i = 0; i < len_A; i++) {

        // we will clear the temporary string
```

```
// every time to accomodate new directory
// name or command.
dir.clear();

// skip all the multiple '/' Eg. "/////"
while (A[i] == '/')
    i++;

// stores directory's name("a", "b" etc.)
// or commands("./".."") into dir
while (i < len_A && A[i] != '/') {
    dir.push_back(A[i]);
    i++;
}

// if dir has ".." just pop the topmost
// element if the stack is not empty
// otherwise ignore.
if (dir.compare("..") == 0) {
    if (!st.empty())
        st.pop();
}

// if dir has "." then simply continue
// with the process.
else if (dir.compare(".") == 0)
    continue;

// pushes if it encounters directory's
// name("a", "b").
else if (dir.length() != 0)
    st.push(dir);
}

// a temporary stack (st1) which will contain
// the reverse of original stack(st).
stack<string> st1;
while (!st.empty()) {
    st1.push(st.top());
    st.pop();
}

// the st1 will contain the actual res.
while (!st1.empty()) {
    string temp = st1.top();

    // if it's the last element no need
    // to append "/"
```

```
        if (st1.size() != 1)
            res.append(temp + "/");
        else
            res.append(temp);

        st1.pop();
    }

    return res;
}

// Driver code.
int main()
{
    // absolute path which we have to simplify.
    string str("/a/./b/../../../../c/");
    string res = simplify(str);
    cout << res;
    return 0;
}
```

Output:

/c

Time Complexity $O(\text{length of string})$.

Source

<https://www.geeksforgeeks.org/simplify-directory-path-unix-like/>

Chapter 104

Sort a stack using a temporary stack

Sort a stack using a temporary stack - GeeksforGeeks

Given a stack of integers, sort it in ascending order using another temporary stack.

Examples:

Input : [34, 3, 31, 98, 92, 23]
Output : [3, 23, 31, 34, 92, 98]

Input : [3, 5, 1, 4, 2, 8]
Output : [1, 2, 3, 4, 5, 8]

We follow this algorithm.

1. Create a temporary stack say **tmpStack**.
2. While input stack is NOT empty do this:
 - Pop an element from input stack call it **temp**
 - while temporary stack is NOT empty and top of temporary stack is greater than temp,
pop from temporary stack and push it to the input stack
 - push **temp** in temporary stack
3. The sorted numbers are in tmpStack

Here is a dry run of above pseudo code.

input: [34, 3, 31, 98, 92, 23]

```
Element taken out: 23
input: [34, 3, 31, 98, 92]
tmpStack: [23]

Element taken out: 92
input: [34, 3, 31, 98]
tmpStack: [23, 92]

Element taken out: 98
input: [34, 3, 31]
tmpStack: [23, 92, 98]

Element taken out: 31
input: [34, 3, 98, 92]
tmpStack: [23, 31]

Element taken out: 92
input: [34, 3, 98]
tmpStack: [23, 31, 92]

Element taken out: 98
input: [34, 3]
tmpStack: [23, 31, 92, 98]

Element taken out: 3
input: [34, 98, 92, 31, 23]
tmpStack: [3]

Element taken out: 23
input: [34, 98, 92, 31]
tmpStack: [3, 23]

Element taken out: 31
input: [34, 98, 92]
tmpStack: [3, 23, 31]

Element taken out: 92
input: [34, 98]
tmpStack: [3, 23, 31, 92]

Element taken out: 98
input: [34]
tmpStack: [3, 23, 31, 92, 98]

Element taken out: 34
input: [98, 92]
tmpStack: [3, 23, 31, 34]
```

```
Element taken out: 92
input: [98]
tmpStack: [3, 23, 31, 34, 92]

Element taken out: 98
input: []
tmpStack: [3, 23, 31, 34, 92, 98]

final sorted list: [3, 23, 31, 34, 92, 98]
```

C++

```
// C++ program to sort a stack using an
// auxiliary stack.
#include <bits/stdc++.h>
using namespace std;

// This function return the sorted stack
stack<int> sortStack(stack<int> &input)
{
    stack<int> tmpStack;

    while (!input.empty())
    {
        // pop out the first element
        int tmp = input.top();
        input.pop();

        // while temporary stack is not empty and top
        // of stack is greater than temp
        while (!tmpStack.empty() && tmpStack.top() > tmp)
        {
            // pop from temporary stack and push
            // it to the input stack
            input.push(tmpStack.top());
            tmpStack.pop();
        }

        // push temp in temporary of stack
        tmpStack.push(tmp);
    }

    return tmpStack;
}

// main function
```

```
int main()
{
    stack<int> input;
    input.push(34);
    input.push(3);
    input.push(31);
    input.push(98);
    input.push(92);
    input.push(23);

    // This is the temporary stack
    stack<int> tmpStack = sortStack(input);
    cout << "Sorted numbers are:\n";

    while (!tmpStack.empty())
    {
        cout << tmpStack.top() << " ";
        tmpStack.pop();
    }
}
```

Java

```
// Java program to sort a stack using
// a auxiliary stack.
import java.util.*;

class SortStack
{
    // This function return the sorted stack
    public static Stack<Integer> sortstack(Stack<Integer>
                                           input)
    {
        Stack<Integer> tmpStack = new Stack<Integer>();
        while(!input.isEmpty())
        {
            // pop out the first element
            int tmp = input.pop();

            // while temporary stack is not empty and
            // top of stack is greater than tmp
            while(!tmpStack.isEmpty() && tmpStack.peek()
                  > tmp)
            {
                // pop from temporary stack and
                // push it to the input stack
                input.push(tmpStack.pop());
            }
        }
    }
}
```

```
        // push temp in temporary of stack
        tmpStack.push(tmp);
    }
    return tmpStack;
}

// Driver Code
public static void main(String args[])
{
    Stack<Integer> input = new Stack<Integer>();
    input.add(34);
    input.add(3);
    input.add(31);
    input.add(98);
    input.add(92);
    input.add(23);

    // This is the temporary stack
    Stack<Integer> tmpStack=sortstack(input);
    System.out.println("Sorted numbers are:");

    while (!tmpStack.empty())
    {
        System.out.print(tmpStack.pop()+" ");
    }
}

// This code is contributed by Danish Kaleem
```

Python3

```
# Python program to sort a
# stack using auxiliary stack.

# This function return the sorted stack
def sortStack ( stack ):
    tmpStack = createStack()
    while(isEmpty(stack) == False):

        # pop out the first element
        tmp = top(stack)
        pop(stack)

        # while temporary stack is not
        # empty and top of stack is
        # greater than temp
        while(isEmpty(tmpStack) == False and
```



```
        int(top(tmpStack)) > int(tmp)):

        # pop from temporary stack and
        # push it to the input stack
        push(stack,top(tmpStack))
        pop(tmpStack)

        # push temp in tempory of stack
        push(tmpStack,tmp)

    return tmpStack

# Below is a complete running
# program for testing above
# function.

# Function to create a stack.
# It initializes size of stack
# as 0
def createStack():
    stack = []
    return stack

# Function to check if
# the stack is empty
def isEmpty( stack ):
    return len(stack) == 0

# Function to push an
# item to stack
def push( stack, item ):
    stack.append( item )

# Function to get top
# item of stack
def top( stack ):
    p = len(stack)
    return stack[p-1]

# Function to pop an
# item from stack
def pop( stack ):

    # If stack is empty
    # then error
    if(isEmpty( stack )):
        print("Stack Underflow ")
        exit(1)
```

```
        return stack.pop()

# Function to print the stack
def prints(stack):
    for i in range(len(stack)-1, -1, -1):
        print(stack[i], end = ' ')
    print()

# Driver Code
stack = createStack()
push( stack, str(34) )
push( stack, str(3) )
push( stack, str(31) )
push( stack, str(98) )
push( stack, str(92) )
push( stack, str(23) )

print("Sorted numbers are: ")
sortedst = sortStack ( stack )
prints(sortedst)

# This code is contributed by
# Prasad Kshirsagar
```

Output:

```
Sorted numbers are:
98 92 34 31 23 3
```

Microsoft

Improved By : [programmer2k17](#), [Prasad_Kshirsagar](#)

Source

<https://www.geeksforgeeks.org/sort-stack-using-temporary-stack/>

Chapter 105

Sort a stack using recursion

Sort a stack using recursion - GeeksforGeeks

Given a stack, sort it using recursion. Use of any loop constructs like while, for..etc is not allowed. We can only use the following ADT functions on Stack S:

```
is_empty(S) : Tests whether stack is empty or not.
push(S)      : Adds new element to the stack.
pop(S)       : Removes top element from the stack.
top(S)       : Returns value of the top element. Note that this
              function does not remove element from the stack.
```

Example:

Input: -3

This problem is mainly a variant of Reverse stack using recursion.

The idea of the solution is to hold all values in Function Call Stack until the stack becomes empty.

Algorithm

We can use below algorithm to sort stack elements:

```
sortStack(stack S)
```

```
if stack is not empty:
    temp = pop(S);
    sortStack(S);
    sortedInsert(S, temp);
```

Below algorithm is to insert element in sorted order:

```
sortedInsert(Stack S, element)
    if stack is empty OR element > top element
        push(S, elem)
    else
        temp = pop(S)
        sortedInsert(S, element)
        push(S, temp)
```

Illustration:

Let given stack be

-3

Let us illustrate sorting of stack using above example:

First pop all the elements from the stack and store popped element in variable 'temp'. After pop

```
temp = -3    --> stack frame #1
temp = 14    --> stack frame #2
temp = 18    --> stack frame #3
temp = -5    --> stack frame #4
temp = 30    --> stack frame #5
```

Now stack is empty and 'insert_in_sorted_order()' function is called and it inserts 30 (from stack frame #5) at the bottom of the stack. Now stack looks like below:

30

Now next element i.e. -5 (from stack frame #4) is picked. Since $-5 < 30$, -5 is inserted at the top

30 -5

Next 18 (from stack frame #3) is picked. Since $18 < 30$, 18 is inserted below 30. Now stack becomes:

30 18
-5

Next 14 (from stack frame #2) is picked. Since $14 < 30$ and $14 < 18$, it is inserted below 18. Now stack becomes:

```
30    14
-5
```

Now -3 (from stack frame #1) is picked, as $-3 < 30$ and $-3 < 18$ and $-3 < 14$, it is inserted below 14. Now stack becomes:

```
30    -3
-5
```

Implementation:

Below is C and Java implementation of above algorithm.

C

```
// C program to sort a stack using recursion
#include <stdio.h>
#include <stdlib.h>

// Stack is represented using linked list
struct stack
{
    int data;
    struct stack *next;
};

// Utility function to initialize stack
void initStack(struct stack **s)
{
    *s = NULL;
}

// Utility function to check if stack is empty
int isEmpty(struct stack *s)
{
    if (s == NULL)
        return 1;
    return 0;
}

// Utility function to push an item to stack
```

```
void push(struct stack **s, int x)
{
    struct stack *p = (struct stack *)malloc(sizeof(*p));

    if (p == NULL)
    {
        fprintf(stderr, "Memory allocation failed.\n");
        return;
    }

    p->data = x;
    p->next = *s;
    *s = p;
}

// Utility function to remove an item from stack
int pop(struct stack **s)
{
    int x;
    struct stack *temp;

    x = (*s)->data;
    temp = *s;
    (*s) = (*s)->next;
    free(temp);

    return x;
}

// Function to find top item
int top(struct stack *s)
{
    return (s->data);
}

// Recursive function to insert an item x in sorted way
void sortedInsert(struct stack **s, int x)
{
    // Base case: Either stack is empty or newly inserted
    // item is greater than top (more than all existing)
    if (isEmpty(*s) || x > top(*s))
    {
        push(s, x);
        return;
    }

    // If top is greater, remove the top item and recur
    int temp = pop(s);
```

```
sortedInsert(s, x);

// Put back the top item removed earlier
push(s, temp);
}

// Function to sort stack
void sortStack(struct stack **s)
{
    // If stack is not empty
    if (!isEmpty(*s))
    {
        // Remove the top item
        int x = pop(s);

        // Sort remaining stack
        sortStack(s);

        // Push the top item back in sorted stack
        sortedInsert(s, x);
    }
}

// Utility function to print contents of stack
void printStack(struct stack *s)
{
    while (s)
    {
        printf("%d ", s->data);
        s = s->next;
    }
    printf("\n");
}

// Driver Program
int main(void)
{
    struct stack *top;

    initStack(&top);
    push(&top, 30);
    push(&top, -5);
    push(&top, 18);
    push(&top, 14);
    push(&top, -3);

    printf("Stack elements before sorting:\n");
    printStack(top);
}
```

```
    sortStack(&top);
    printf("\n\n");

    printf("Stack elements after sorting:\n");
    printStack(top);

    return 0;
}
```

Java

```
// Java program to sort a Stack using recursion
// Note that here predefined Stack class is used
// for stack operation

import java.util.ListIterator;
import java.util.Stack;

class Test
{
    // Recursive Method to insert an item x in sorted way
    static void sortedInsert(Stack<Integer> s, int x)
    {
        // Base case: Either stack is empty or newly inserted
        // item is greater than top (more than all existing)
        if (s.isEmpty() || x > s.peek())
        {
            s.push(x);
            return;
        }

        // If top is greater, remove the top item and recur
        int temp = s.pop();
        sortedInsert(s, x);

        // Put back the top item removed earlier
        s.push(temp);
    }

    // Method to sort stack
    static void sortStack(Stack<Integer> s)
    {
        // If stack is not empty
        if (!s.isEmpty())
        {
            // Remove the top item
            int x = s.pop();
```



```
        // Sort remaining stack
        sortStack(s);

        // Push the top item back in sorted stack
        sortedInsert(s, x);
    }
}

// Utility Method to print contents of stack
static void printStack(Stack<Integer> s)
{
    ListIterator<Integer> lt = s.listIterator();

    // forwarding
    while(lt.hasNext())
        lt.next();

    // printing from top to bottom
    while(lt.hasPrevious())
        System.out.print(lt.previous()+" ");
}

// Driver method
public static void main(String[] args)
{
    Stack<Integer> s = new Stack<>();
    s.push(30);
    s.push(-5);
    s.push(18);
    s.push(14);
    s.push(-3);

    System.out.println("Stack elements before sorting: ");
    printStack(s);

    sortStack(s);

    System.out.println(" \n\nStack elements after sorting:");
    printStack(s);
}
}
```

Output:

Stack elements before sorting:

-3 14 18 -5 30

Stack elements after sorting:

30 18 14 -3 -5

Exercise: Modify above code to reverse stack in descending order.

This article is contributed by **Narendra Kangralkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/sort-a-stack-using-recursion/>

Chapter 106

Sort string of characters using Stack

Sort string of characters using Stack - GeeksforGeeks

Given a string of characters. The task is to write a program to print the characters of this string in sorted order using stack.

Examples:

Input: str = "geeksforgeeks"

Output: eeeefggkkorss

Input: str = "hello395world216"

Output: 123569dehlloorw

Approach:

- Initialize two stacks, one **stack** and other **tempstack**.
- Insert the first character of the string in the **stack**.
-
- if the i^{th} character is greater than or equal to the top element of the stack, then push the element.
- if the i^{th} character is not greater, then push all the elements of the **stack** into **tempstack**, and then push the character into the **stack**. After this, push all the greater elements of **tempstack** to **stack**.

Print the all elements of the **stack** in reverse order when the iteration is completed.

Below is the implementation of the above approach:

```
# Python program to sort
# string of characters using stack

# function to print the characters
# in sorted order
def printSorted(s, l):

    # primary stack
    stack = []

    # secondary stack
    tempstack = []

    # append first character
    stack.append(s[0])

    # iterate for all character in string
    for i in range(1, l):

        # i-th character ASCII
        a = ord(s[i])

        # stack's top element ASCII
        b = ord(stack[-1])

        # if greater or equal to top element
        # then push to stack
        if((a-b)>= 1 or (a == b)):
            stack.append(s[i])

        # if smaller, then push all element
        # to the temporary stack
        elif((b-a)>= 1):

            # push all greater elements
            while((b-a)>= 1):

                # push operation
                tempstack.append(stack.pop())

            # push till the stack is not-empty
            if(len(stack)>0):
                b = ord(stack[-1])
            else:
                break

        # push the i-th character
```

```
        stack.append(s[i])

        # push the tempstack back to stack
        while(len(tempstack)>0):
            stack.append(tempstack.pop())

        # print the stack in reverse order
        print(''.join(stack))

# Driver Code
s = "geeksforgeeks"
l = len(s)
printSorted(s, l)
```

Output:

eeeefggkkrss

An efficient approach using hashing has been implemented in [this post](#)

Source

<https://www.geeksforgeeks.org/sort-string-of-characters-using-stack/>

Chapter 107

Sorting array using Stacks

Sorting array using Stacks - GeeksforGeeks

Given an array of elements, task is to sort these elements using stack.

Prerequisites : [Stacks](#)

Examples :

Input : 8 5 7 1 9 12 10
Output : 1 5 7 8 9 10 12
Explanation :
Output is sorted element set

Input : 7 4 10 20 2 5 9 1
Output : 1 2 4 5 7 9 10 20

We basically use [Sort a stack using a temporary stack](#). Then we put sorted stack elements back to array.

C++

```
// C++ program to sort an array using stack
#include <bits/stdc++.h>
using namespace std;

// This function return the sorted stack
stack<int> sortStack(stack<int> input)
{
    stack<int> tmpStack;

    while (!input.empty())
```

```
{
    // pop out the first element
    int tmp = input.top();
    input.pop();

    // while temporary stack is not empty
    // and top of stack is smaller than temp
    while (!tmpStack.empty() &&
           tmpStack.top() < tmp)
    {
        // pop from temporary stack and
        // push it to the input stack
        input.push(tmpStack.top());
        tmpStack.pop();
    }

    // push temp in tempory of stack
    tmpStack.push(tmp);
}

return tmpStack;
}

void sortArrayUsingStacks(int arr[], int n)
{
    // Push array elements to stack
    stack<int> input;
    for (int i=0; i<n; i++)
        input.push(arr[i]);

    // Sort the temporary stack
    stack<int> tmpStack = sortStack(input);

    // Put stack elements in arrp[]
    for (int i=0; i<n; i++)
    {
        arr[i] = tmpStack.top();
        tmpStack.pop();
    }
}

// main function
int main()
{
    int arr[] = {10, 5, 15, 45};
    int n = sizeof(arr)/sizeof(arr[0]);

    sortArrayUsingStacks(arr, n);
}
```

```
    for (int i=0; i<n; i++)
        cout << arr[i] << " ";

    return 0;
}
```

Java

```
// Java program to sort an
// array using stack
import java.io.*;
import java.util.*;

class GFG
{
    // This function return
    // the sorted stack
    static Stack<Integer> sortStack(Stack<Integer> input)
    {
        Stack<Integer> tmpStack =
            new Stack<Integer>();

        while (!input.empty())
        {
            // pop out the
            // first element
            int tmp = input.peek();
            input.pop();

            // while temporary stack is
            // not empty and top of stack
            // is smaller than temp
            while (!tmpStack.empty() &&
                tmpStack.peek() < tmp)
            {
                // pop from temporary
                // stack and push it
                // to the input stack
                input.push(tmpStack.peek());
                tmpStack.pop();
            }

            // push temp in
            // tempory of stack
            tmpStack.push(tmp);
        }
    }
}
```



```
        return tmpStack;
    }

    static void sortArrayUsingStacks(int []arr,
                                     int n)
    {
        // push array elements
        // to stack
        Stack<Integer> input =
            new Stack<Integer>();
        for (int i = 0; i < n; i++)
            input.push(arr[i]);

        // Sort the temporary stack
        Stack<Integer> tmpStack =
            sortStack(input);

        // Put stack elements
        // in arrp[]
        for (int i = 0; i < n; i++)
        {
            arr[i] = tmpStack.peek();
            tmpStack.pop();
        }
    }

    // Driver Code
    public static void main(String args[])
    {
        int []arr = {10, 5, 15, 45};
        int n = arr.length;

        sortArrayUsingStacks(arr, n);

        for (int i = 0; i < n; i++)
            System.out.print(arr[i] + " ");
    }

    }

    // This code is contributed by
    // Manish Shaw(manishshaw1)
```

C#

```
// C# program to sort an
// array using stack
using System;
using System.Collections.Generic;
```

```
class GFG
{
    // This function return
    // the sorted stack
    static Stack<int> sortStack(Stack<int> input)
    {
        Stack<int> tmpStack = new Stack<int>();

        while (input.Count != 0)
        {
            // pop out the
            // first element
            int tmp = input.Peek();
            input.Pop();

            // while temporary stack is
            // not empty and top of stack
            // is smaller than temp
            while (tmpStack.Count != 0 &&
                tmpStack.Peek() < tmp)
            {
                // pop from temporary
                // stack and push it
                // to the input stack
                input.Push(tmpStack.Peek());
                tmpStack.Pop();
            }

            // push temp in
            // tempory of stack
            tmpStack.Push(tmp);
        }

        return tmpStack;
    }

    static void sortArrayUsingStacks(int []arr,
                                     int n)
    {
        // Push array elements
        // to stack
        Stack<int> input = new Stack<int>();
        for (int i = 0; i<n; i++)
            input.Push(arr[i]);

        // Sort the temporary stack
        Stack<int> tmpStack = sortStack(input);
    }
}
```

```
        // Put stack elements in arrp[]
        for (int i = 0; i < n; i++)
        {
            arr[i] = tmpStack.Peek();
            tmpStack.Pop();
        }
    }

    // Driver Code
    static void Main()
    {
        int []arr = new int[] {10, 5,
                               15, 45};
        int n = arr.Length;

        sortArrayUsingStacks(arr, n);

        for (int i = 0; i < n; i++)
            Console.Write(arr[i] + " ");
    }
}

// This code is contributed by
// Manish Shaw(manishshaw1)
```

Output:

5 10 15 45

Time Complexity : $O(n^2)$

Improved By : [manishshaw1](#)

Source

<https://www.geeksforgeeks.org/sorting-array-using-stacks/>

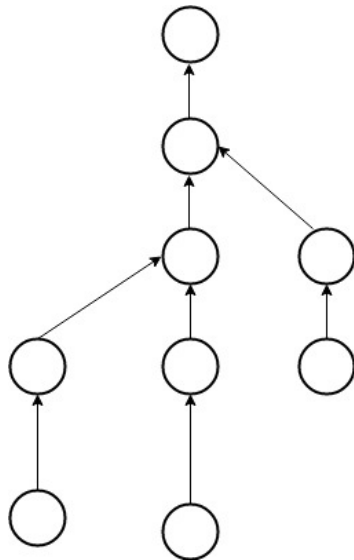
Chapter 108

Spaghetti Stack

Spaghetti Stack - GeeksforGeeks

Spaghetti stack

A spaghetti stack is an N-ary tree data structure in which child nodes have pointers to the parent nodes (but not vice-versa)



Spaghetti stack structure is used in situations when records are dynamically pushed and popped onto a stack as execution progresses, but references to the popped records remain in use. Following are some applications of Spaghetti Stack.

Compilers for languages such as C create a spaghetti stack as it opens and closes symbol tables representing block scopes. When a new block scope is opened, a symbol table is pushed onto a stack. When the closing curly brace is encountered, the scope is closed and the symbol table is popped. But that symbol table is remembered, rather than destroyed. And of course it remembers its higher level “parent” symbol table and so on.

Spaghetti Stacks are also used to implement [Disjoint-set data structure](#).

Sources:

http://en.wikipedia.org/wiki/Spaghetti_stack

Source

<https://www.geeksforgeeks.org/g-fact-87/>

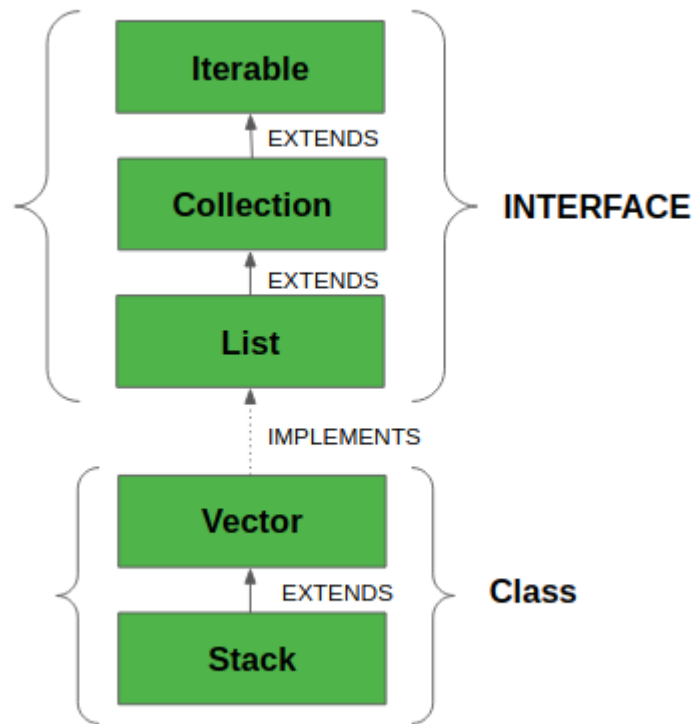
Chapter 109

Stack Class in Java

Stack Class in Java - GeeksforGeeks

Java Collection framework provides a Stack class which models and implements Stack data structure. The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek. The class can also be said to extend Vector and treats the class as a stack with the five mentioned functions. The class can also be referred to as the subclass of Vector.

This diagram shows the hierarchy of Stack class:



The class supports one *default constructor* **Stack()** which is used to *create an empty stack*. Below program shows few basic operations provided by the Stack class:

```
// Java code for stack implementation

import java.io.*;
import java.util.*;

class Test
{
    // Pushing element on the top of the stack
    static void stack_push(Stack<Integer> stack)
    {
        for(int i = 0; i < 5; i++)
        {
            stack.push(i);
        }
    }

    // Popping element from the top of the stack
    static void stack_pop(Stack<Integer> stack)
    {
        System.out.println("Pop :");
    }
}
```

```
        for(int i = 0; i < 5; i++)
        {
            Integer y = (Integer) stack.pop();
            System.out.println(y);
        }

// Displaying element on the top of the stack
static void stack_peek(Stack<Integer> stack)
{
    Integer element = (Integer) stack.peek();
    System.out.println("Element on stack top : " + element);
}

// Searching element in the stack
static void stack_search(Stack<Integer> stack, int element)
{
    Integer pos = (Integer) stack.search(element);

    if(pos == -1)
        System.out.println("Element not found");
    else
        System.out.println("Element is found at position " + pos);
}

public static void main (String[] args)
{
    Stack<Integer> stack = new Stack<Integer>();

    stack_push(stack);
    stack_pop(stack);
    stack_push(stack);
    stack_peek(stack);
    stack_search(stack, 2);
    stack_search(stack, 6);
}
}
```

Output:

```
Pop :
4
3
2
1
```



```
0
Element on stack top : 4
Element is found at position 3
Element not found
```

Methods in Stack class

1. **Object push(*Object element*)** : Pushes an element on the top of the stack.
2. **Object pop()** : Removes and returns the top element of the stack. An 'EmptyStack-Exception' exception is thrown if we call pop() when the invoking stack is empty.
3. **Object peek()** : Returns the element on the top of the stack, but does not remove it.
4. **boolean empty()** : It returns true if nothing is on the top of the stack. Else, returns false.
5. **int search(*Object element*)** : It determines whether an object exists in the stack. If the element is found, it returns the position of the element from the top of the stack. Else, it returns -1.

This article is contributed by Mehak Narang.

Improved By : [Chinmoy Lenka](#)

Source

<https://www.geeksforgeeks.org/stack-class-in-java/>

Chapter 110

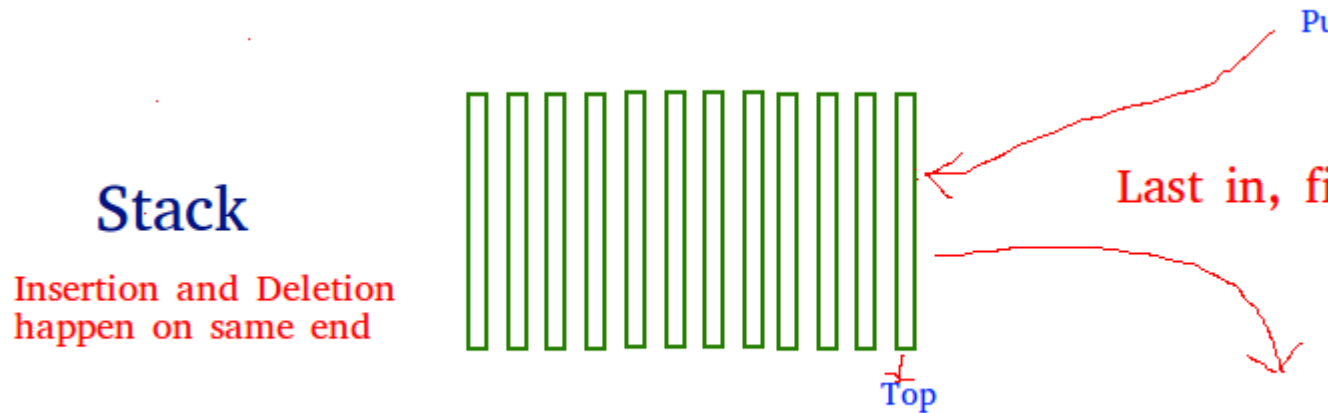
Stack Data Structure (Introduction and Program)

Stack Data Structure (Introduction and Program) - GeeksforGeeks

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).

Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.



How to understand a stack practically?

There are many real life examples of stack. Consider the simple example of plates stacked over one another in canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.

Time Complexities of operations on stack:

push(), pop(), isEmpty() and peek() all take $O(1)$ time. We do not run any loop in any of these operations.

Applications of stack:

- [Balancing of symbols](#)
- [Infix to Postfix](#) /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like [Tower of Hanoi](#), [tree traversals](#), [stock span problem](#), [histogram problem](#).
- Other applications can be Backtracking, [Knight tour problem](#), [rat in a maze](#), [N queen problem](#) and [sudoku solver](#)
- In Graph Algorithms like [Topological Sorting](#) and [Strongly Connected Components](#)

Implementation:

There are two ways to implement a stack:

- Using array
- Using linked list

Implementing Stack using Arrays

C++

```
/* C++ program to implement basic stack
operations */
#include<bits/stdc++.h>

using namespace std;

#define MAX 1000

class Stack
{
    int top;
public:
    int a[MAX];    //Maximum size of Stack

    Stack() { top = -1; }
    bool push(int x);
    int pop();
    bool isEmpty();
};

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

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

```
bool Stack::isEmpty()
{
    return (top < 0);
}

// Driver program to test above functions
int main()
{
    struct Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout<<s.pop() << " Popped from stack\n";

    return 0;
}
```

C

```
// C program for array implementation of stack
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a stack
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};

// function to create a stack of given capacity. It initializes size of
// stack as 0
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*) malloc(stack->capacity * sizeof(int));
    return stack;
}

// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{
    return stack->top == stack->capacity - 1; }
}
```

```
// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{   return stack->top == -1;   }

// Function to add an item to stack. It increases top by 1
void push(struct Stack* stack, int item)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
    printf("%d pushed to stack\n", item);
}

// Function to remove an item from stack. It decreases top by 1
int pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}

// Driver program to test above functions
int main()
{
    struct Stack* stack = createStack(100);

    push(stack, 10);
    push(stack, 20);
    push(stack, 30);

    printf("%d popped from stack\n", pop(stack));

    return 0;
}
```

Java

```
/* Java program to implement basic stack
operations */
class Stack
{
    static final int MAX = 1000;
    int top;
    int a[] = new int[MAX]; // Maximum size of Stack

    boolean isEmpty()
    {
        return (top < 0);
    }
}
```

```
Stack()
{
    top = -1;
}

boolean push(int x)
{
    if (top >= (MAX-1))
    {
        System.out.println("Stack Overflow");
        return false;
    }
    else
    {
        a[++top] = x;
        System.out.println(x + " pushed into stack");
        return true;
    }
}

int pop()
{
    if (top < 0)
    {
        System.out.println("Stack Underflow");
        return 0;
    }
    else
    {
        int x = a[top--];
        return x;
    }
}

// Driver code
class Main
{
    public static void main(String args[])
    {
        Stack s = new Stack();
        s.push(10);
        s.push(20);
        s.push(30);
        System.out.println(s.pop() + " Popped from stack");
    }
}
```

Python

```
# Python program for implementation of stack

# import maxsize from sys module
# Used to return -infinite when stack is empty
from sys import maxsize

# Function to create a stack. It initializes size of stack as 0
def createStack():
    stack = []
    return stack

# Stack is empty when stack size is 0
def isEmpty(stack):
    return len(stack) == 0

# Function to add an item to stack. It increases size by 1
def push(stack, item):
    stack.append(item)
    print(item + " pushed to stack ")

# Function to remove an item from stack. It decreases size by 1
def pop(stack):
    if (isEmpty(stack)):
        return str(-maxsize -1) #return minus infinite

    return stack.pop()

# Driver program to test above functions
stack = createStack()
push(stack, str(10))
push(stack, str(20))
push(stack, str(30))
print(pop(stack) + " popped from stack")
```

C#

```
// C# program to implement basic stack
// operations
using System;

namespace ImplementStack
{
    class Stack
    {
        private int[] ele;
```



```
private int top;
private int max;
public Stack(int size)
{
    ele = new int[size]; //Maximum size of Stack
    top = -1;
    max = size;
}

public void push(int item)
{
    if (top == max-1)
    {
        Console.WriteLine("Stack Overflow");
        return;
    }
    else
    {
        ele[++top] = item;
    }
}

public int pop()
{
    if(top == -1)
    {
        Console.WriteLine("Stack is Empty");
        return -1;
    }
    else
    {
        Console.WriteLine("{0} popped from stack ", ele[top]);
        return ele[top--];
    }
}

public void printStack()
{
    if (top == -1)
    {
        Console.WriteLine("Stack is Empty");
        return;
    }
    else
    {
        for (int i = 0; i <= top; i++)
        {
            Console.WriteLine("{0} pushed into stack", ele[i]);
        }
    }
}
```

```
        }
    }
}

// Driver program to test above functions
class Program
{
    static void Main()
    {
        Stack p = new Stack(5);

        p.push(10);
        p.push(20);
        p.push(30);
        p.printStack();
        p.pop();
    }
}
```

Pros: Easy to implement. Memory is saved as pointers are not involved.

Cons: It is not dynamic. It doesn't grow and shrink depending on needs at runtime.

Output :

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
30 popped from stack
```

Implementing Stack using Linked List

C

```
// C program for linked list implementation of stack
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a stack
struct StackNode
{
    int data;
    struct StackNode* next;
};

struct StackNode* newNode(int data)
```

```
{
    struct StackNode* stackNode =
        (struct StackNode*) malloc(sizeof(struct StackNode));
    stackNode->data = data;
    stackNode->next = NULL;
    return stackNode;
}

int isEmpty(struct StackNode *root)
{
    return !root;
}

void push(struct StackNode** root, int data)
{
    struct StackNode* stackNode = newNode(data);
    stackNode->next = *root;
    *root = stackNode;
    printf("%d pushed to stack\n", data);
}

int pop(struct StackNode** root)
{
    if (isEmpty(*root))
        return INT_MIN;
    struct StackNode* temp = *root;
    *root = (*root)->next;
    int popped = temp->data;
    free(temp);

    return popped;
}

int peek(struct StackNode* root)
{
    if (isEmpty(root))
        return INT_MIN;
    return root->data;
}

int main()
{
    struct StackNode* root = NULL;

    push(&root, 10);
    push(&root, 20);
    push(&root, 30);
}
```

```
    printf("%d popped from stack\n", pop(&root));

    printf("Top element is %d\n", peek(root));

    return 0;
}
```

Python

```
# Python program for linked list implementation of stack
```

```
# Class to represent a node
```

```
class StackNode:
```

```
    # Constructor to initialize a node
```

```
    def __init__(self, data):
```

```
        self.data = data
```

```
        self.next = None
```

```
class Stack:
```

```
    # Constructor to initialize the root of linked list
```

```
    def __init__(self):
```

```
        self.root = None
```

```
    def isEmpty(self):
```

```
        return True if self.root is None else False
```

```
    def push(self, data):
```

```
        newNode = StackNode(data)
```

```
        newNode.next = self.root
```

```
        self.root = newNode
```

```
        print "%d pushed to stack" %(data)
```

```
    def pop(self):
```

```
        if (self.isEmpty()):
```

```
            return float("-inf")
```

```
        temp = self.root
```

```
        self.root = self.root.next
```

```
        popped = temp.data
```

```
        return popped
```

```
    def peek(self):
```

```
        if self.isEmpty():
```

```
            return float("-inf")
```

```
        return self.root.data
```

```
# Driver program to test above class
```

```
stack = Stack()
stack.push(10)
stack.push(20)
stack.push(30)

print "%d popped from stack" %(stack.pop())
print "Top element is %d " %(stack.peak())

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
10 pushed to stack
20 pushed to stack
30 pushed to stack
30 popped from stack
Top element is 20
```

Pros: The linked list implementation of stack can grow and shrink according to the needs at runtime.

Cons: Requires extra memory due to involvement of pointers.

We will cover the implementation of applications of stack in separate posts.

[Stack Set -2 \(Infix to Postfix\)](#)

Quiz: [Stack Questions](#)

References:

http://en.wikipedia.org/wiki/Stack_%28abstract_data_type%29#Problem_Description

Improved By : [SUNILKUMAR19](#), [SoumikMondal](#)

Source

<https://www.geeksforgeeks.org/stack-data-structure-introduction-program/>

Chapter 111

Stack Permutations (Check if an array is stack permutation of other)

Stack Permutations (Check if an array is stack permutation of other) - GeeksforGeeks

A **stack permutation** is a permutation of objects in the given input queue which is done by transferring elements from input queue to the output queue with the help of a stack and the built-in push and pop functions.

The well defined rules are:

1. Only dequeue from the input queue.
2. Use inbuilt push, pop functions in the single stack.
3. Stack and input queue must be empty at the end.
4. Only enqueue to the output queue.

There are a huge number of permutations possible using a stack for a single input queue. Given two arrays, both of unique elements. One represents the input queue and the other represents the output queue. Our task is to check if the given output is possible through stack permutation.

Examples:

```
Input : First array: 1, 2, 3
        Second array: 2, 1, 3
Output : Yes
Procedure:
push 1 from input to stack
push 2 from input to stack
```

```
pop 2 from stack to output
pop 1 from stack to output
push 3 from input to stack
pop 3 from stack to output
```

```
Input : First array: 1, 2, 3
        Second array: 3, 1, 2
Output : Not Possible
```

The idea to do this is we will try to convert the input queue to output queue using a stack, if we are able to do so then the queue is permutable otherwise not.

Below is the step by step algorithm to do this:

1. Continuously pop elements from the input queue and check if it is equal to the top of output queue or not, if it is not equal to the top of output queue then we will push the element to stack.
2. Once we find an element in input queue such the top of input queue is equal to top of output queue, we will pop a single element from both input and output queues, and compare the top of stack and top of output queue now. If top of both stack and output queue are equal then pop element from both stack and output queue. If not equal, go to step 1.
3. Repeat above two steps until the input queue becomes empty. At the end if both of the input queue and stack are empty then the input queue is permutable otherwise not.

Below is C++ implementation of above idea:

```
// Given two arrays, check if one array is
// stack permutation of other.
#include<bits/stdc++.h>
using namespace std;

// function to check if input queue is
// permutable to output queue
bool checkStackPermutation(int ip[], int op[], int n)
{
    // Input queue
    queue<int> input;
    for (int i=0;i<n;i++)
        input.push(ip[i]);

    // output queue
    queue<int> output;
    for (int i=0;i<n;i++)
        output.push(op[i]);
```

```
// stack to be used for permutation
stack <int> tempStack;
while (!input.empty())
{
    int ele = input.front();
    input.pop();
    if (ele == output.front())
    {
        output.pop();
        while (!tempStack.empty())
        {
            if (tempStack.top() == output.front())
            {
                tempStack.pop();
                output.pop();
            }
            else
                break;
        }
    }
    else
        tempStack.push(ele);
}

// If after processing, both input queue and
// stack are empty then the input queue is
// permutable otherwise not.
return (input.empty() && tempStack.empty());
}

// Driver program to test above function
int main()
{
    // Input Queue
    int input[] = {1, 2, 3};

    // Output Queue
    int output[] = {2, 1, 3};

    int n = 3;

    if (checkStackPermutation(input, output, n))
        cout << "Yes";
    else
        cout << "Not Possible";
    return 0;
}
```


Output:

Yes

Source

<https://www.geeksforgeeks.org/stack-permutations-check-if-an-array-is-stack-permutation-of-other/>

Chapter 112

Stack and Queue in Python using queue Module

Stack and Queue in Python using queue Module - GeeksforGeeks

A simple python List can act as queue and stack as well. Queue mechanism is used widely and for many purposes in daily life. A queue follows FIFO rule(First In First Out) and is used in programming for sorting and for many more things. Python provides Class queue as a module which has to be generally created in languages such as C/C++ and Java.

1. Creating a FIFO Queue

```
// Initialize queue  
Syntax: queue.Queue(maxsize)  
  
// Insert Element  
Syntax: Queue.put(data)  
  
// Get And remove the element  
Syntax: Queue.get()
```

Initializes a variable to a maximum size of maxsize. A maxsize of zero '0' means a infinite queue. This Queue follows FIFO rule. This module also has a LIFO Queue, which is basically a Stack. Data is inserted into Queue using put() and the end. get() takes data out from the front of the Queue. Note that Both put() and get() take 2 more parameters, optional flags, block and timeout.

```
import queue  
  
# From class queue, Queue is  
# created as an object Now L
```

```
# is Queue of a maximum
# capacity of 20
L = queue.Queue(maxsize=20)

# Data is inserted into Queue
# using put() Data is inserted
# at the end
L.put(5)
L.put(9)
L.put(1)
L.put(7)

# get() takes data out from
# the Queue from the head
# of the Queue
print(L.get())
print(L.get())
print(L.get())
print(L.get())
```

Output:

```
5
9
1
7
```

2. UnderFlow and OverFlow

When we try to add data into a Queue above is maxsize, it is called OverFlow(Queue Full) and when we try removing an element from an empty, it's called Underflow. put() and get() do not give error upon Underflow and Overflow, but goes into an infinite loop.

```
import queue

L = queue.Queue(maxsize=6)

# qsize() give the maxsize
# of the Queue
print(L.qsize())

L.put(5)
L.put(9)
L.put(1)
L.put(7)

# Return Boolean for Full
```

```
# Queue
print("Full: ", L.full())

L.put(9)
L.put(10)
print("Full: ", L.full())

print(L.get())
print(L.get())
print(L.get())

# Return Boolean for Empty
# Queue
print("Empty: ", L.empty())

print(L.get())
print(L.get())
print(L.get())

print("Empty: ", L.empty())
print("Full: ", L.full())

# This would result into Infinite
# Loop as the Queue is empty.
# print(L.get())
```

Output:

```
0
Full:  False
Full:  True
5
9
1
Empty:  False
7
9
10
Empty:  True
Full:  False
```

3. Stack

This module queue also provides LIFO Queue which technically works as a Stack.

```
import queue
```

```
L = queue.LifoQueue(maxsize=6)

# qsize() give the maxsize of
# the Queue
print(L.qsize())

# Data Inserted as 5->9->1->7,
# same as Queue
L.put(5)
L.put(9)
L.put(1)
L.put(7)
L.put(9)
L.put(10)
print("Full: ", L.full())
print("Size: ", L.qsize())

# Data will be accessed in the
# reverse order Reverse of that
# of Queue
print(L.get())
print(L.get())
print(L.get())
print(L.get())
print(L.get())
print("Empty: ", L.empty())
```

Output:

```
0
Full:  True
Size:  6
10
9
7
1
9
Empty:  False
```

Reference:

<https://docs.python.org/3/library/asyncio-queue.html>

Source

<https://www.geeksforgeeks.org/stack-queue-python-using-module-queue/>

Chapter 113

Stack | Set 2 (Infix to Postfix)

Stack | Set 2 (Infix to Postfix) - GeeksforGeeks

Prerequisite – [Stack | Set 1 \(Introduction\)](#)

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

Postfix expression: 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.

Consider the below expression: a op1 b op2 c op3 d

If op1 = +, op2 = *, op3 = +

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.

The corresponding expression in postfix form is: abc*+d+. The postfix expressions can be evaluated easily using a stack. We will cover postfix expression evaluation in a separate post.

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.

Following is C implementation of the above algorithm

C

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Stack type
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};

// Stack Operations
struct Stack* createStack( unsigned capacity )
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));

    if (!stack)
        return NULL;

    stack->top = -1;
    stack->capacity = capacity;

    stack->array = (int*) malloc(stack->capacity * sizeof(int));

    if (!stack->array)
        return NULL;
    return stack;
}

int isEmpty(struct Stack* stack)
{
    return stack->top == -1 ;
}

char peek(struct Stack* stack)
{
    return stack->array[stack->top];
}

char pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--] ;
    return '$';
}
```

```
}
void push(struct Stack* stack, char op)
{
    stack->array[++stack->top] = op;
}

// A utility function to check if the given character is operand
int isOperand(char ch)
{
    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}

// A utility function to return precedence of a given operator
// Higher returned value means higher precedence
int Prec(char ch)
{
    switch (ch)
    {
        case '+':
        case '-':
            return 1;

        case '*':
        case '/':
            return 2;

        case '^':
            return 3;
    }
    return -1;
}

// The main function that converts given infix expression
// to postfix expression.
int infixToPostfix(char* exp)
{
    int i, k;

    // Create a stack of capacity equal to expression size
    struct Stack* stack = createStack(strlen(exp));
    if(!stack) // See if stack was created successfully
        return -1 ;

    for (i = 0, k = -1; exp[i]; ++i)
    {
        // If the scanned character is an operand, add it to output.
```



```

    if (isOperand(exp[i]))
        exp[++k] = exp[i];

    // If the scanned character is an '(', push it to the stack.
    else if (exp[i] == '(')
        push(stack, exp[i]);

    // If the scanned character is an ')', pop and output from the stack
    // until an '(' is encountered.
    else if (exp[i] == ')')
    {
        while (!isEmpty(stack) && peek(stack) != '(')
            exp[++k] = pop(stack);
        if (!isEmpty(stack) && peek(stack) != '(')
            return -1; // invalid expression
        else
            pop(stack);
    }
    else // an operator is encountered
    {
        while (!isEmpty(stack) && Prec(exp[i]) <= Prec(peek(stack)))
            exp[++k] = pop(stack);
        push(stack, exp[i]);
    }
}

// pop all the operators from the stack
while (!isEmpty(stack))
    exp[++k] = pop(stack );

exp[++k] = '\0';
printf( "%sn", exp );
}

```

```

// Driver program to test above functions
int main()
{
    char exp[] = "a+b*(c^d-e)^(f+g*h)-i";
    infixToPostfix(exp);
    return 0;
}

```

C++

```

/* C++ implementation to convert infix expression to postfix*/
// Note that here we use std::stack for Stack operations
#include<bits/stdc++.h>

```

```

using namespace std;

//Function to return precedence of operators
int prec(char c)
{
    if(c == '^')
        return 3;
    else if(c == '*' || c == '/')
        return 2;
    else if(c == '+' || c == '-')
        return 1;
    else
        return -1;
}

// The main function to convert infix expression
//to postfix expression
void infixToPostfix(string s)
{
    std::stack<char> st;
    st.push('N');
    int l = s.length();
    string ns;
    for(int i = 0; i < l; i++)
    {
        // If the scanned character is an operand, add it to output string.
        if((s[i] >= 'a' && s[i] <= 'z') || (s[i] >= 'A' && s[i] <= 'Z'))
            ns+=s[i];

        // If the scanned character is an '(', push it to the stack.
        else if(s[i] == '(')

            st.push('(');

        // If the scanned character is an ')', pop and to output string from the stack
        // until an '(' is encountered.
        else if(s[i] == ')')
        {
            while(st.top() != 'N' && st.top() != '(')
            {
                char c = st.top();
                st.pop();
                ns += c;
            }
            if(st.top() == '(')
            {
                char c = st.top();
                st.pop();
            }
        }
    }
    ns += st.top();
    st.pop();
    cout << ns << endl;
}

```

```

        }
    }

    //If an operator is scanned
    else{
        while(st.top() != 'N' && prec(s[i]) <= prec(st.top()))
        {
            char c = st.top();
            st.pop();
            ns += c;
        }
        st.push(s[i]);
    }

}

//Pop all the remaining elements from the stack
while(st.top() != 'N')
{
    char c = st.top();
    st.pop();
    ns += c;
}

cout << ns << endl;

}

//Driver program to test above functions
int main()
{
    string exp = "a+b*(c^d-e)^(f+g*h)-i";
    infixToPostfix(exp);
    return 0;
}

// This code is contributed by Gautam Singh

```

Java

```

/* Java implementation to convert infix expression to postfix*/
// Note that here we use Stack class for Stack operations

import java.util.Stack;

class Test
{
    // A utility function to return precedence of a given operator
    // Higher returned value means higher precedence
    static int Prec(char ch)

```

```

{
    switch (ch)
    {
        case '+':
        case '-':
            return 1;

        case '*':
        case '/':
            return 2;

        case '^':
            return 3;
    }
    return -1;
}

// The main method that converts given infix expression
// to postfix expression.
static String infixToPostfix(String exp)
{
    // initializing empty String for result
    String result = new String("");

    // initializing empty stack
    Stack<Character> stack = new Stack<>();

    for (int i = 0; i<exp.length(); ++i)
    {
        char c = exp.charAt(i);

        // If the scanned character is an operand, add it to output.
        if (Character.isLetterOrDigit(c))
            result += c;

        // If the scanned character is an '(', push it to the stack.
        else if (c == '(')
            stack.push(c);

        // If the scanned character is an ')', pop and output from the stack
        // until an '(' is encountered.
        else if (c == ')')
        {
            while (!stack.isEmpty() && stack.peek() != '(')
                result += stack.pop();

            if (!stack.isEmpty() && stack.peek() != '(')
                return "Invalid Expression"; // invalid expression
        }
    }
}

```

```

        else
            stack.pop();
    }
    else // an operator is encountered
    {
        while (!stack.isEmpty() && Prec(c) <= Prec(stack.peek()))
            result += stack.pop();
        stack.push(c);
    }

}

// pop all the operators from the stack
while (!stack.isEmpty())
    result += stack.pop();

return result;
}

// Driver method
public static void main(String[] args)
{
    String exp = "a+b*(c^d-e)^(f+g*h)-i";
    System.out.println(infixToPostfix(exp));
}
}

```

Python

```

# Python program to convert infix expression to postfix

# Class to convert the expression
class Conversion:

    # Constructor to initialize the class variables
    def __init__(self, capacity):
        self.top = -1
        self.capacity = capacity
        # This array is used a stack
        self.array = []
        # Precedence setting
        self.output = []
        self.precedence = {'+':1, '-':1, '*':2, '/':2, '^':3}

    # check if the stack is empty
    def isEmpty(self):
        return True if self.top == -1 else False

```

```
# Return the value of the top of the stack
def peek(self):
    return self.array[-1]

# Pop the element from the stack
def pop(self):
    if not self.isEmpty():
        self.top -= 1
        return self.array.pop()
    else:
        return "$"

# Push the element to the stack
def push(self, op):
    self.top += 1
    self.array.append(op)

# A utility function to check is the given character
# is operand
def isOperand(self, ch):
    return ch.isalpha()

# Check if the precedence of operator is strictly
# less than top of stack or not
def notGreater(self, i):
    try:
        a = self.precedence[i]
        b = self.precedence[self.peek()]
        return True if a <= b else False
    except KeyError:
        return False

# The main function that converts given infix expression
# to postfix expression
def infixToPostfix(self, exp):

    # Iterate over the expression for conversion
    for i in exp:
        # If the character is an operand,
        # add it to output
        if self.isOperand(i):
            self.output.append(i)

        # If the character is an '(', push it to stack
        elif i == '(':
            self.push(i)

        # If the scanned character is an ')', pop and
```

```

        # output from the stack until and '(' is found
        elif i == ')':
            while( (not self.isEmpty()) and self.peek() != '('):
                a = self.pop()
                self.output.append(a)
            if (not self.isEmpty() and self.peek() != '('):
                return -1
            else:
                self.pop()

        # An operator is encountered
        else:
            while(not self.isEmpty() and self.notGreater(i)):
                self.output.append(self.pop())
            self.push(i)

        # pop all the operator from the stack
        while not self.isEmpty():
            self.output.append(self.pop())

        print "".join(self.output)

# Driver program to test above function
exp = "a+b*(c^d-e)^(f+g*h)-i"
obj = Conversion(len(exp))
obj.infixToPostfix(exp)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```
abcd^e-fgh*+^*+i-
```

Quiz: [Stack Questions](#)

Source

<https://www.geeksforgeeks.org/stack-set-2-infix-to-postfix/>

Chapter 114

Stack | Set 3 (Reverse a string using stack)

Stack | Set 3 (Reverse a string using stack) - GeeksforGeeks

Given a string, reverse it using stack. For example “GeeksQuiz” should be converted to “ziuQskeeG”.

Following is simple algorithm to reverse a string using stack.

- 1) Create an empty stack.
- 2) One by one push all characters of string to stack.
- 3) One by one pop all characters from stack and put them back to string.

Following is C++ implementation of program

```
// C++ program to reverse a string using stack
#include<bits/stdc++.h>

using namespace std;

//Class to implement Stack
class Stack
{
    private:
        char * a;
    public:
        int t;
        void pop();
        void push(char b);
}
```



```
bool empty();
Stack(int size)
{
    a= new char[size];
    t=-1;
}
};

// Class to print out element
void Stack::pop()
{
    if(empty())
    {
        cout<<"Empty Stack"<<endl;
        return;
    }

    cout<<a[t];
    t--;
}

//Class to insert element
void Stack::push(char b)
{
    a[++t]=b;
}

//To check if stack is empty or not
bool Stack::empty()
{
    return t<0;
}

//Function to reverse string
void reverse(Stack k)
{
    while(!k.empty())
    {
        k.pop();
    }
}

// Driver code
int main() {

    Stack block(5);
    block.push('h');
    block.push('e');
    block.push('l');
```

```

    block.push('l');
    block.push('o');
    reverse(block);
    return 0;
}

```

Following programs implements above algorithm.

C

```

// C program to reverse a string using stack
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a stack
struct Stack
{
    int top;
    unsigned capacity;
    char* array;
};

// function to create a stack of given
// capacity. It initializes size of stack as 0
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (char*) malloc(stack->capacity * sizeof(char));
    return stack;
}

// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{ return stack->top == stack->capacity - 1; }

// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{ return stack->top == -1; }

// Function to add an item to stack.
// It increases top by 1
void push(struct Stack* stack, char item)
{

```

```
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
}

// Function to remove an item from stack.
// It decreases top by 1
char pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}

// A stack based function to reverse a string
void reverse(char str[])
{
    // Create a stack of capacity
    //equal to length of string
    int n = strlen(str);
    struct Stack* stack = createStack(n);

    // Push all characters of string to stack
    int i;
    for (i = 0; i < n; i++)
        push(stack, str[i]);

    // Pop all characters of string and
    // put them back to str
    for (i = 0; i < n; i++)
        str[i] = pop(stack);
}

// Driver program to test above functions
int main()
{
    char str[] = "GeeksQuiz";

    reverse(str);
    printf("Reversed string is %s", str);

    return 0;
}
```

Java

```
/* Java program to reverse
String using Stack */
```

```
import java.util.*;

//stack
class Stack
{
    int size;
    int top;
    char[] a;

    //function to check if stack is empty
    boolean isEmpty()
    {
        return (top < 0);
    }

    Stack(int n)
    {
        top = -1;
        size = n;
        a = new char[size];
    }

    //function to push element in Stack
    boolean push(char x)
    {
        if (top >= size)
        {
            System.out.println("Stack Overflow");
            return false;
        }
        else
        {
            a[++top] = x;
            return true;
        }
    }

    //function to pop element from stack
    char pop()
    {
        if (top < 0)
        {
            System.out.println("Stack Underflow");
            return 0;
        }
        else
        {
            return a[top--];
        }
    }
}
```

```
        char x = a[top--];
        return x;
    }
}

// Driver code
class Main
{
    //function to reverse the string
    public static void reverse(StringBuffer str)
    {
        // Create a stack of capacity
        // equal to length of string
        int n = str.length();
        Stack obj = new Stack(n);

        // Push all characters of string
        // to stack
        int i;
        for (i = 0; i < n; i++)
            obj.push(str.charAt(i));

        // Pop all characters of string
        // and put them back to str
        for (i = 0; i < n; i++)
        {
            char ch = obj.pop();
            str.setCharAt(i, ch);
        }
    }

    //driver function
    public static void main(String args[])
    {
        //create a new string
        StringBuffer s= new StringBuffer("GeeksQuiz");

        //call reverse method
        reverse(s);

        //print the reversed string
        System.out.println("Reversed string is " + s);
    }
}
```

Python

```
# Python program to reverse a string using stack

# Function to create an empty stack.
# It initializes size of stack as 0
def createStack():
    stack=[]
    return stack

# Function to determine the size of the stack
def size(stack):
    return len(stack)

# Stack is empty if the size is 0
def isEmpty(stack):
    if size(stack) == 0:
        return true

# Function to add an item to stack .
# It increases size by 1
def push(stack,item):
    stack.append(item)

#Function to remove an item from stack.
# It decreases size by 1
def pop(stack):
    if isEmpty(stack): return
    return stack.pop()

# A stack based function to reverse a string
def reverse(string):
    n = len(string)

    # Create a empty stack
    stack = createStack()

    # Push all characters of string to stack
    for i in range(0,n,1):
        push(stack,string[i])

    # Making the string empty since all
    #characters are saved in stack
    string=""

    # Pop all characters of string and
    # put them back to string
    for i in range(0,n,1):
        string+=pop(stack)
```

```
    return string

# Driver program to test above functions
string="GeeksQuiz"
string = reverse(string)
print("Reversed string is " + string)

# This code is contributed by Sunny Karira
```

Output:

Reversed string is ziuQskeeG

Time Complexity: $O(n)$ where n is number of characters in stack.

Auxiliary Space: $O(n)$ for stack.

A string can also be reversed without using any auxiliary space. Following C and Python programs to implement reverse without using stack.

C

```
// C program to reverse a string without using stack
#include <stdio.h>
#include <string.h>

// A utility function to swap two characters
void swap(char *a, char *b)
{
    char temp = *a;
    *a = *b;
    *b = temp;
}

// A stack based function to reverse a string
void reverse(char str[])
{
    // get size of string
    int n = strlen(str), i;

    for (i = 0; i < n/2; i++)
        swap(&str[i], &str[n-i-1]);
}

// Driver program to test above functions
int main()
{
    char str[] = "abc";
```

```
    reverse(str);
    printf("Reversed string is %s", str);

    return 0;
}
```

Python

```
# Python program to reverse a string without stack

# Function to reverse a string
def reverse(string):
    string = string[::-1]
    return string

# Driver program to test above functions
string = "abc"
string = reverse(string)
print("Reversed string is " + string)

# This code is contributed by Sunny Karira
```

Output:

Reversed string is cba

Improved By : [krikti](#)

Source

<https://www.geeksforgeeks.org/stack-set-3-reverse-string-using-stack/>

Chapter 115

Stack | Set 4 (Evaluation of Postfix Expression)

Stack | Set 4 (Evaluation of Postfix Expression) - GeeksforGeeks

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. We have discussed [infix to postfix conversion](#). In this post, evaluation of postfix expressions is discussed.

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 an 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

Example:

Let the given expression be “2 3 1 * + 9 -“. We scan all elements one by one.

- 1) Scan ‘2’, it’s a number, so push it to stack. Stack contains ‘2’
- 2) Scan ‘3’, again a number, push it to stack, stack now contains ‘2 3’ (from bottom to top)
- 3) Scan ‘1’, again a number, push it to stack, stack now contains ‘2 3 1’
- 4) Scan ‘*’, it’s an operator, pop two operands from stack, apply the * operator on operands, we get 3*1 which results in 3. We push the result ‘3’ to stack. Stack now becomes ‘2 3’.
- 5) Scan ‘+’, it’s an operator, pop two operands from stack, apply the + operator on operands, we get 3 + 2 which results in 5. We push the result ‘5’ to stack. Stack now becomes ‘5’.
- 6) Scan ‘9’, it’s a number, we push it to the stack. Stack now becomes ‘5 9’.
- 7) Scan ‘-’, it’s an operator, pop two operands from stack, apply the – operator on operands, we get 5 – 9 which results in -4. We push the result ‘-4’ to stack. Stack now becomes ‘-4’.
- 8) There are no more elements to scan, we return the top element from stack (which is the only element left in stack).

Below is the implementation of above algorithm.

C

```
// C program to evaluate value of a postfix expression
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

// Stack type
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};

// Stack Operations
struct Stack* createStack( unsigned capacity )
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));

    if (!stack) return NULL;

    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (int*) malloc(stack->capacity * sizeof(int));

    if (!stack->array) return NULL;

    return stack;
}

int isEmpty(struct Stack* stack)
{
    return stack->top == -1 ;
}

char peek(struct Stack* stack)
{
    return stack->array[stack->top];
}

char pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--] ;
    return '$';
}
```

```
void push(struct Stack* stack, char op)
{
    stack->array[++stack->top] = op;
}

// The main function that returns value of a given postfix expression
int evaluatePostfix(char* exp)
{
    // Create a stack of capacity equal to expression size
    struct Stack* stack = createStack(strlen(exp));
    int i;

    // See if stack was created successfully
    if (!stack) return -1;

    // Scan all characters one by one
    for (i = 0; exp[i]; ++i)
    {
        // If the scanned character is an operand (number here),
        // push it to the stack.
        if (isdigit(exp[i]))
            push(stack, exp[i] - '0');

        // If the scanned character is an operator, pop two
        // elements from stack apply the operator
        else
        {
            int val1 = pop(stack);
            int val2 = pop(stack);
            switch (exp[i])
            {
                case '+': push(stack, val2 + val1); break;
                case '-': push(stack, val2 - val1); break;
                case '*': push(stack, val2 * val1); break;
                case '/': push(stack, val2/val1); break;
            }
        }
    }
    return pop(stack);
}

// Driver program to test above functions
int main()
{
    char exp[] = "231*+9-";
    printf ("Value of %s is %d", exp, evaluatePostfix(exp));
}
```

```
    return 0;
}
```

Java

```
// Java program to evaluate value of a postfix expression

import java.util.Stack;

public class Test
{
    // Method to evaluate value of a postfix expression
    static int evaluatePostfix(String exp)
    {
        //create a stack
        Stack<Integer> stack=new Stack<>();

        // Scan all characters one by one
        for(int i=0;i<exp.length();i++)
        {
            char c=exp.charAt(i);

            // If the scanned character is an operand (number here),
            // push it to the stack.
            if(Character.isDigit(c))
                stack.push(c - '0');

            // If the scanned character is an operator, pop two
            // elements from stack apply the operator
            else
            {
                int val1 = stack.pop();
                int val2 = stack.pop();

                switch(c)
                {
                    case '+':
                        stack.push(val2+val1);
                        break;

                    case '-':
                        stack.push(val2- val1);
                        break;

                    case '/':
                        stack.push(val2/val1);
                        break;
                }
            }
        }
    }
}
```

```
                case '*':
                    stack.push(val2*val1);
                    break;
            }
        }
    }
    return stack.pop();
}

// Driver program to test above functions
public static void main(String[] args)
{
    String exp="231*+9-";
    System.out.println(evaluatePostfix(exp));
}
// Contributed by Sumit Ghosh
```

Python

```
# Python program to evaluate value of a postfix expression

# Class to convert the expression
class Evaluate:

    # Constructor to initialize the class variables
    def __init__(self, capacity):
        self.top = -1
        self.capacity = capacity
        # This array is used a stack
        self.array = []

    # check if the stack is empty
    def isEmpty(self):
        return True if self.top == -1 else False

    # Return the value of the top of the stack
    def peek(self):
        return self.array[-1]

    # Pop the element from the stack
    def pop(self):
        if not self.isEmpty():
            self.top -= 1
            return self.array.pop()
        else:
            return "$"

    # Push the element into the stack
    def push(self, val):
        if self.top < self.capacity - 1:
            self.array.append(val)
            self.top += 1
        else:
            return "$"

    # Evaluate the postfix expression
    def evaluate(self, exp):
        for i in range(len(exp)):
            ch = exp[i]

            if ch.isdigit():
                # If the character is a digit,
                # push the digit to stack
                val = int(ch)
                self.push(val)
            elif ch in ['+', '-', '*', '/']:
                # If the character is an operator,
                # pop two elements from the stack
                # apply the operator on them
                val1 = self.pop()
                val2 = self.pop()
                res = 0
                if ch == '+':
                    res = val1 + val2
                elif ch == '-':
                    res = val2 - val1
                elif ch == '*':
                    res = val1 * val2
                elif ch == '/':
                    res = val2 / val1
                self.push(res)
            else:
                return "$"

        return self.array[-1]
```

```
# Push the element to the stack
def push(self, op):
    self.top += 1
    self.array.append(op)

# The main function that converts given infix expression
# to postfix expression
def evaluatePostfix(self, exp):

    # Iterate over the expression for conversion
    for i in exp:

        # If the scanned character is an operand
        # (number here) push it to the stack
        if i.isdigit():
            self.push(i)

        # If the scanned character is an operator,
        # pop two elements from stack and apply it.
        else:
            val1 = self.pop()
            val2 = self.pop()
            self.push(str(eval(val2 + i + val1)))

    return int(self.pop())

# Driver program to test above function
exp = "231*+9-"
obj = Evaluate(len(exp))
print "Value of %s is %d" %(exp, obj.evaluatePostfix(exp))

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

Value of 231*+9- is -4

Time complexity of evaluation algorithm is $O(n)$ where n is number of characters in input expression.

There are following limitations of above implementation.

- 1) It supports only 4 binary operators '+', '*', '-', and '/'. It can be extended for more operators by adding more switch cases.
- 2) The allowed operands are only single digit operands. The program can be extended

for multiple digits by adding a separator like space between all elements (operators and operands) of given expression.

Below given is the extended program which allows operands having multiple digits.

C

```
// C program to evaluate value of a postfix
// expression having multiple digit operands
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

// Stack type
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};

// Stack Operations
struct Stack* createStack( unsigned capacity )
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));

    if (!stack) return NULL;

    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (int*) malloc(stack->capacity * sizeof(int));

    if (!stack->array) return NULL;

    return stack;
}

int isEmpty(struct Stack* stack)
{
    return stack->top == -1 ;
}

int peek(struct Stack* stack)
{
    return stack->array[stack->top];
}
```

```
int pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--] ;
    return '$';
}

void push(struct Stack* stack,int op)
{
    stack->array[++stack->top] = op;
}

// The main function that returns value
// of a given postfix expression
int evaluatePostfix(char* exp)
{
    // Create a stack of capacity equal to expression size
    struct Stack* stack = createStack(strlen(exp));
    int i;

    // See if stack was created successfully
    if (!stack) return -1;

    // Scan all characters one by one
    for (i = 0; exp[i]; ++i)
    {
        //if the character is blank space then continue
        if(exp[i]==' ')continue;

        // If the scanned character is an
        // operand (number here),extract the full number
        // Push it to the stack.
        else if (isdigit(exp[i]))
        {
            int num=0;

            //extract full number
            while(isdigit(exp[i]))
            {
                num=num*10 + (int)(exp[i]-'0');
                i++;
            }
            i--;

            //push the element in the stack
            push(stack,num);
        }
    }
}
```



```
// If the scanned character is an operator, pop two
// elements from stack apply the operator
else
{
    int val1 = pop(stack);
    int val2 = pop(stack);

    switch (exp[i])
    {
        case '+': push(stack, val2 + val1); break;
        case '-': push(stack, val2 - val1); break;
        case '*': push(stack, val2 * val1); break;
        case '/': push(stack, val2/val1); break;
    }
}
return pop(stack);
}

// Driver program to test above functions
int main()
{
    char exp[] = "100 200 + 2 / 5 * 7 +";
    printf ("%d", evaluatePostfix(exp));
    return 0;
}

// This code is contributed by Arnab Kundu
```

Java

```
// Java program to evaluate value of a postfix
// expression having multiple digit operands
import java.util.Stack;

class Test1
{
    // Method to evaluate value of a postfix expression
    static int evaluatePostfix(String exp)
    {
        //create a stack
        Stack<Integer> stack = new Stack<>();

        // Scan all characters one by one
        for(int i = 0; i < exp.length(); i++)
        {
```

```
char c = exp.charAt(i);

if(c == ' ')
    continue;

// If the scanned character is an operand
// (number here),extract the number
// Push it to the stack.
else if(Character.isDigit(c))
{
    int n = 0;

    //extract the characters and store it in num
    while(Character.isDigit(c))
    {
        n = n*10 + (int)(c-'0');
        i++;
        c = exp.charAt(i);
    }
    i--;

    //push the number in stack
    stack.push(n);
}

// If the scanned character is an operator, pop two
// elements from stack apply the operator
else
{
    int val1 = stack.pop();
    int val2 = stack.pop();

    switch(c)
    {
        case '+':
            stack.push(val2+val1);
            break;

        case '-':
            stack.push(val2- val1);
            break;

        case '/':
            stack.push(val2/val1);
            break;

        case '*':
            stack.push(val2*val1);
```

```
                break;
            }
        }
    }
    return stack.pop();
}

// Driver program to test above functions
public static void main(String[] args)
{
    String exp = "100 200 + 2 / 5 * 7 +";
    System.out.println(evaluatePostfix(exp));
}

// This code is contributed by Arnab Kundu
```

Output :

757

References:

<http://www.cs.nthu.edu.tw/~wkhon/ds/ds10/tutorial/tutorial2.pdf>

Improved By : [andrew1234](#)

Source

<https://www.geeksforgeeks.org/stack-set-4-evaluation-postfix-expression/>

Chapter 116

Sudo Placement[1.3] | Playing with Stacks

Sudo Placement[1.3] | Playing with Stacks - GeeksforGeeks

You are given 3 stacks, A(Input Stack), B(Auxiliary Stack) and C(Output Stack). Initially stack A contains numbers from 1 to N, you need to transfer all the numbers from stack A to stack C in sorted order i.e in the end, the stack C should have smallest element at the bottom and largest at top. You can use stack B i.e at any time you can push/pop elements to stack B also. At the end stack A, B should be empty.

Examples:

Input: A = {4, 3, 1, 2, 5}

Output: Yes 7

Input: A = {3, 4, 1, 2, 5}

Output: No

Approach: Iterate from the bottom of the given stack. Initialize *required* as the bottom most element in stackC at the end i.e., 1. Follow the given below algorithm to solve the above problem.

- if the stack element is equal to the required element, then the number of transfers will be one which is the count of transferring from A to C.
- if it is not equal to the required element, then check if it is possible to transfer it by comparing it with the topmost element in the stack.
 1. If the topmost element in stackC is greater than the stackA[i] element, then it is not possible to transfer it in a sorted way,
 2. else push the element to stackC and increment transfer.
- Iterate in the stackC and pop out the top most element until it is equal to the required and increment required and transfer in every steps.

Below is the implementation of the above approach:

```
// C++ program for
// Sudo Placement | playing with stacks
#include <bits/stdc++.h>
using namespace std;

// Function to check if it is possible
// count the number of steps
void countSteps(int sa[], int n)
{
    // Another stack
    stack<int> sc;

    // variables to count transfers
    int required = 1, transfer = 0;

    // iterate in the stack in reverse order
    for (int i = 0; i < n; i++) {

        // if the last element has to be
        // inserted by removing elements
        // then count the number of steps
        if (sa[i] == required) {
            required++;
            transfer++;
        }
        else {
            // if stack is not empty and top element
            // is smaller than current element
            if (!sc.empty() && sc.top() < sa[i]) {
                cout << "NO";
                return;
            }
            // push into stack and count operation
            else {

                sc.push(sa[i]);
                transfer++;
            }
        }
        // stack not empty, then pop the top element
        // pop out all elements till is it equal to required
        while (!sc.empty() && sc.top() == required) {
            required++;
            sc.pop();
            transfer++;
        }
    }
}
```

```
        }
    }

    // print the steps
    cout << "YES " << transfer;
}

// Driver Code
int main()
{
    int sa[] = { 4, 3, 1, 2, 5 };
    int n = sizeof(sa) / sizeof(sa[0]);
    countSteps(sa, n);
    return 0;
}
```

Output:

YES 7

Source

<https://www.geeksforgeeks.org/sudo-placement-playing-with-stacks/>

Chapter 117

Sudo Placement[1.3] | Stack Design

Sudo Placement[1.3] | Stack Design - GeeksforGeeks

Given q number of queries, you need to perform operations on the stack. Queries are of three types 1, 2 and 3. If the operation is to push (1) then push the elements, if the operations is to pop (2) then pop the element and if it is Top (3), then print the element at the top of stack (If stack is empty, print “-1”, without quotes).

Examples:

```
Input: Queries = 6
                 3
                 1 5
                 1 6
                 1 7
                 2
                 3
```

```
Output: -1
        6
```

The first query is to print top, but since the stack is empty, so we print -1. Next three queries are to push 5, 6, and 7, so we pushed them on a stack. Next query is pop, so we popped 7 from a stack. Final query is to print the top, so 6 is there at the top and thus printed.

Approach: [Stack](#) can be used to perform the given operation. If the input for a query is 1, then take another input and push the element into the stack using [push\(\)](#) function. If the input for a query is 2, then pop the element using [pop\(\)](#) function in a stack. If the input for a query is 3, then print the top element using [top\(\)](#) function.

Below is the implemenatation of the above aprpoch:

```
// C++ program for
// Sudo-Placement | Stack Design
#include <bits/stdc++.h>
using namespace std;

stack<int> s;

// function to perform type-1 operation
void _push(int n)
{
    s.push(n);
}

// function to perform type-2 operation
void _pop()
{
    s.pop();
}

// function to perform type-3 operation
void print()
{
    // if the stack is not empty
    if (!s.empty())
        cout << s.top() << endl;
    else
        cout << -1 << endl;
}

// Driver Code
int main()
{
    // 1st query
    print();
    // 2nd query
    _push(5);

    // 3rd query
    _push(6);

    // 4th query
    _push(7);

    // 5th query
    _pop();

    // 6th query
    print();
}
```



```
    return 0;  
}
```

Output:

```
-1  
6
```

Source

<https://www.geeksforgeeks.org/sudo-placement1-3-stack-design/>

Chapter 118

The Celebrity Problem

The Celebrity Problem - GeeksforGeeks

*In a party of N people, only one person is known to everyone. Such a person **may be present** in the party, if yes, (s)he doesn't know anyone in the party. We can only ask questions like “**does A know B?** “. Find the stranger (celebrity) in minimum number of questions.*

We can describe the problem input as an array of numbers/characters representing persons in the party. We also have a hypothetical function *HaveAcquaintance(A, B)* which returns *true* if A knows B, *false* otherwise. How can we solve the problem.

We measure the complexity in terms of calls made to *HaveAcquaintance()*.

Method 1 (Graph)

We can model the solution using graphs. Initialize indegree and outdegree of every vertex as 0. If A knows B, draw a directed edge from A to B, increase indegree of B and outdegree of A by 1. Construct all possible edges of the graph for every possible pair $[i, j]$. We have N_{C_2} pairs. If celebrity is present in the party, we will have one sink node in the graph with outdegree of zero, and indegree of $N-1$. We can find the sink node in (N) time, but the overall complexity is $O(N^2)$ as we need to construct the graph first.

Method 2 (Recursion)

We can decompose the problem into combination of smaller instances. Say, if we know celebrity of $N-1$ persons, can we extend the solution to N ? We have two possibilities, Celebrity($N-1$) may know N , or N already knew Celebrity($N-1$). In the former case, N will be celebrity if N doesn't know anyone else. In the later case we need to check that Celebrity($N-1$) doesn't know N .

Solve the problem of smaller instance during divide step. On the way back, we find the celebrity (if present) from the smaller instance. During combine stage, check whether the returned celebrity is known to everyone and he doesn't know anyone. The recurrence of the recursive decomposition is,

$$T(N) = T(N-1) + O(N)$$

$T(N) = O(N^2)$. You may try writing pseudo code to check your recursion skills.

Method 3 (Using Stack)

The graph construction takes $O(N^2)$ time, it is similar to brute force search. In case of recursion, we reduce the problem instance by not more than one, and also combine step may examine $M-1$ persons (M – instance size).

We have following observation based on elimination technique (Refer *Polya's How to Solve It* book).

- If A knows B, then A can't be celebrity. Discard A, and *B may be celebrity*.
- If A doesn't know B, then B can't be celebrity. Discard B, and *A may be celebrity*.
- Repeat above two steps till we left with only one person.
- Ensure the remained person is celebrity. (Why do we need this step?)

We can use stack to verify celebrity.

1. Push all the celebrities into a stack.
2. Pop off top two persons from the stack, discard one person based on return status of *HaveAcquaintance(A, B)*.
3. Push the remained person onto stack.
4. Repeat step 2 and 3 until only one person remains in the stack.
5. Check the remained person in stack doesn't have acquaintance with anyone else.

We will discard N elements utmost (Why?). If the celebrity is present in the party, we will call *HaveAcquaintance()* $3(N-1)$ times. Here is code using stack.

C++

```
// C++ program to find celebrity
#include <bits/stdc++.h>
#include <list>
using namespace std;

// Max # of persons in the party
#define N 8

// Person with 2 is celebrity
bool MATRIX[N][N] = {{0, 0, 1, 0},
                     {0, 0, 1, 0},
                     {0, 0, 0, 0},
                     {0, 0, 1, 0}};

bool knows(int a, int b)
{
    return MATRIX[a][b];
}
```

```
// Returns -1 if celebrity
// is not present. If present,
// returns id (value from 0 to n-1).
int findCelebrity(int n)
{
    // Handle trivial
    // case of size = 2

    stack<int> s;

    int C; // Celebrity

    // Push everybody to stack
    for (int i = 0; i < n; i++)
        s.push(i);

    // Extract top 2
    int A = s.top();
    s.pop();
    int B = s.top();
    s.pop();

    // Find a potential celebrity
    while (s.size() > 1)
    {
        if (knows(A, B))
        {
            A = s.top();
            s.pop();
        }
        else
        {
            B = s.top();
            s.pop();
        }
    }

    // Potential candidate?
    C = s.top();
    s.pop();

    // Last candidate was not
    // examined, it leads one
    // excess comparison (optimize)
    if (knows(C, B))
        C = B;

    if (knows(C, A))
```

```
        C = A;

        // Check if C is actually
        // a celebrity or not
        for (int i = 0; i < n; i++)
        {
            // If any person doesn't
            // know 'a' or 'a' doesn't
            // know any person, return -1
            if ( (i != C) &&
                (knows(C, i) ||
                 !knows(i, C)) )
                return -1;
        }

        return C;
    }

    // Driver code
    int main()
    {
        int n = 4;
        int id = findCelebrity(n);
        id == -1 ? cout << "No celebrity" :
                  cout << "Celebrity ID " << id;

        return 0;
    }
```

Java

```
// Java program to find celebrity using
// stack data structure

import java.util.Stack;

class GFG
{
    // Person with 2 is celebrity
    static int MATRIX[][] = { { 0, 0, 1, 0 },
                              { 0, 0, 1, 0 },
                              { 0, 0, 0, 0 },
                              { 0, 0, 1, 0 } };

    // Returns true if a knows
    // b, false otherwise
    static boolean knows(int a, int b)
    {
        boolean res = (MATRIX[a][b] == 1) ?
```

```

                                true :
                                false;

    return res;
}

// Returns -1 if celebrity
// is not present. If present,
// returns id (value from 0 to n-1).
static int findCelebrity(int n)
{
    Stack<Integer> st = new Stack<>();
    int c;

    // Step 1 :Push everybody
    // onto stack
    for (int i = 0; i < n; i++)
    {
        st.push(i);
    }

    while (st.size() > 1)
    {
        // Step 2 :Pop off top
        // two persons from the
        // stack, discard one
        // person based on return
        // status of knows(A, B).
        int a = st.pop();
        int b = st.pop();

        // Step 3 : Push the
        // remained person onto stack.
        if (knows(a, b))
        {
            st.push(b);
        }

        else
            st.push(a);
    }

    c = st.pop();

    // Step 5 : Check if the last
    // person is celebrity or not
    for (int i = 0; i < n; i++)
    {
        // If any person doesn't

```

```

        // know 'c' or 'a' doesn't
        // know any person, return -1
        if (i != c && (knows(c, i) ||
                        !knows(i, c)))
            return -1;
    }
    return c;
}

// Driver Code
public static void main(String[] args)
{
    int n = 4;
    int result = findCelebrity(n);
    if (result == -1)
    {
        System.out.println("No Celebrity");
    }
    else
        System.out.println("Celebrity ID " +
                            result);
}

// This code is contributed
// by Rishabh Mahrsee

```

Output :

Celebrity ID 2

Complexity $O(N)$. Total comparisons $3(N-1)$. Try the above code for successful MATRIX $\{\{0, 0, 0, 1\}, \{0, 0, 0, 1\}, \{0, 0, 0, 1\}, \{0, 0, 0, 1\}\}$.

Note: You may think that why do we need a new graph as we already have access to input matrix. Note that the matrix MATRIX used to help the hypothetical function *HaveAcquaintance*(A, B), but never accessed via usual notation MATRIX[i, j]. We have access to the input only through the function *HaveAcquaintance*(A, B). Matrix is just a way to code the solution. We can assume the cost of hypothetical function as $O(1)$.

If still not clear, assume that the function *HaveAcquaintance* accessing information stored in a set of linked lists arranged in levels. List node will have *next* and *nextLevel* pointers. Every level will have N nodes i.e. an N element list, *next* points to next node in the current level list and the *nextLevel* pointer in last node of every list will point to head of next level list. For example the linked list representation of above matrix looks like,

```

L0 0->0->1->0
    |

```

```

L1          0->0->1->0
              |
L2          0->0->1->0
              |
L3          0->0->1->0

```

The function *HaveAcquaintance*(*i*, *j*) will search in the list for *j*-th node in the *i*-th level. Our goal is to minimize calls to *HaveAcquaintance* function.

Method 4 (Using two Pointers)

The idea is to use two pointers, one from start and one from the end. Assume the start person is A, and the end person is B. If A knows B, then A must not be the celebrity. Else, B must not be the celebrity. We will find a celebrity candidate at the end of the loop. Go through each person again and check whether this is the celebrity. Below is C++ implementation.

C++

```

// C++ program to find
// celebrity in O(n) time
// and O(1) extra space
#include <bits/stdc++.h>
using namespace std;

// Max # of persons in the party
#define N 8

// Person with 2 is celebrity
bool MATRIX[N][N] = {{0, 0, 1, 0},
                     {0, 0, 1, 0},
                     {0, 0, 0, 0},
                     {0, 0, 1, 0}};

};

bool knows(int a, int b)
{
    return MATRIX[a][b];
}

// Returns id of celebrity
int findCelebrity(int n)
{
    // Initialize two pointers
    // as two corners
    int a = 0;
    int b = n - 1;

```



```
// Keep moving while
// the two pointers
// don't become same.
while (a < b)
{
    if (knows(a, b))
        a++;
    else
        b--;
}

// Check if a is actually
// a celebrity or not
for (int i = 0; i < n; i++)
{
    // If any person doesn't
    // know 'a' or 'a' doesn't
    // know any person, return -1
    if ( (i != a) &&
         (knows(a, i) ||
          !knows(i, a)) )
        return -1;
}

return a;
}

// Driver code
int main()
{
    int n = 4;
    int id = findCelebrity(n);
    id == -1 ? cout << "No celebrity" :
              cout << "Celebrity ID "
                  << id;

    return 0;
}
```

Java

```
// Java program to find
// celebrity using two
// pointers

class GFG
{
    // Person with 2 is celebrity
    static int MATRIX[][] = { { 0, 0, 1, 0 },
```

```
        { 0, 0, 1, 0 },
        { 0, 0, 0, 0 },
        { 0, 0, 1, 0 } }];

// Returns true if a knows
// b, false otherwise
static boolean knows(int a, int b)
{
    boolean res = (MATRIX[a][b] == 1) ?
                   true :
                   false;

    return res;
}

// Returns -1 if celebrity
// is not present. If present,
// returns id (value from 0 to n-1).
static int findCelebrity(int n)
{
    // Initialize two pointers
    // as two corners
    int a = 0;
    int b = n - 1;

    // Keep moving while
    // the two pointers
    // don't become same.
    while (a < b)
    {
        if (knows(a, b))
            a++;
        else
            b--;
    }

    // Check if a is actually
    // a celebrity or not
    for (int i = 0; i < n; i++)
    {
        // If any person doesn't
        // know 'a' or 'a' doesn't
        // know any person, return -1
        if (i != a && (knows(a, i) ||
                       !knows(i, a)))
            return -1;
    }
    return a;
}
```

```
// Driver Code
public static void main(String[] args)
{
    int n = 4;
    int result = findCelebrity(n);
    if (result == -1)
    {
        System.out.println("No Celebrity");
    }
    else
        System.out.println("Celebrity ID " +
                             result);
}
```

// This code is contributed by Rishabh Mahrsee

C#

```
// C# program to find
// celebrity using two
// pointers
using System;

class GFG
{
    // Person with 2 is celebrity
    static int [,]MATRIX = {{ 0, 0, 1, 0 },
                             { 0, 0, 1, 0 },
                             { 0, 0, 0, 0 },
                             { 0, 0, 1, 0 }};

    // Returns true if a knows
    // b, false otherwise
    static bool knows(int a, int b)
    {
        bool res = (MATRIX[a, b] == 1) ?
                    true :
                    false;

        return res;
    }

    // Returns -1 if celebrity
    // is not present. If present,
    // returns id (value from 0 to n-1).
    static int findCelebrity(int n)
    {
```

```
// Initialize two pointers
// as two corners
int a = 0;
int b = n - 1;

// Keep moving while
// the two pointers
// don't become same.
while (a < b)
{
    if (knows(a, b))
        a++;
    else
        b--;
}

// Check if a is actually
// a celebrity or not
for (int i = 0; i < n; i++)
{
    // If any person doesn't
    // know 'a' or 'a' doesn't
    // know any person, return -1
    if (i != a && (knows(a, i) ||
                    !knows(i, a)))
        return -1;
}
return a;
}

// Driver Code
public static void Main()
{
    int n = 4;
    int result = findCelebrity(n);
    if (result == -1)
    {
        Console.WriteLine("No Celebrity");
    }
    else
        Console.WriteLine("Celebrity ID " +
                           result);
}

// This code is contributed by anuj_67.
```

PHP

```
<?php
// PHP program to find
// celebrity in O(n) time
// and O(1) extra space

// Max # of persons
// in the party $N = 8;

// Person with 2 is celebrity
$MATRIX = array(array(0, 0, 1, 0),
                  array(0, 0, 1, 0),
                  array(0, 0, 0, 0),
                  array(0, 0, 1, 0));

function knows( $a, $b)
{
    global $MATRIX;
    return $MATRIX[$a][$b];
}

// Returns id of celebrity
function findCelebrity( $n)
{
    // Initialize two
    // pointers as two corners
    $a = 0;
    $b = $n - 1;

    // Keep moving while
    // the two pointers
    // don't become same.
    while ($a < $b)
    {
        if (knows($a, $b))
            $a++;
        else
            $b--;
    }

    // Check if a is actually
    // a celebrity or not
    for ( $i = 0; $i < $n; $i++)
    {
        // If any person doesn't
        // know 'a' or 'a' doesn't
        // know any person, return -1
        if ( ($i != $a) and
```

```
        (knows($a, $i) ||
         !knows($i, $a)) )
    return -1;
}

return $a;
}

// Driver code
$n = 4;
$id = findCelebrity($n);
if($id == -1)
echo "No celebrity" ;
else
echo "Celebrity ID " , $id;

// This code is contributed by anuj_67.
?>
```

Output :

Celebrity ID 2

Thanks to Sissi Peng for suggesting this method.

Related Article:

[Number of sink nodes in a graph](#)

Exercises:

1. Write code to find celebrity. Don't use any data structures like graphs, stack, etc... you have access to N and *HaveAcquaintance(int, int)* only.
2. Implement the algorithm using Queues. What is your observation? Compare your solution with [Finding Maximum and Minimum](#) in an array and [Tournament Tree](#). What are minimum number of comparisons do we need (optimal number of calls to *HaveAcquaintance()*)?

— [Venki](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : [vt_m](#)

Source

<https://www.geeksforgeeks.org/the-celebrity-problem/>

Chapter 119

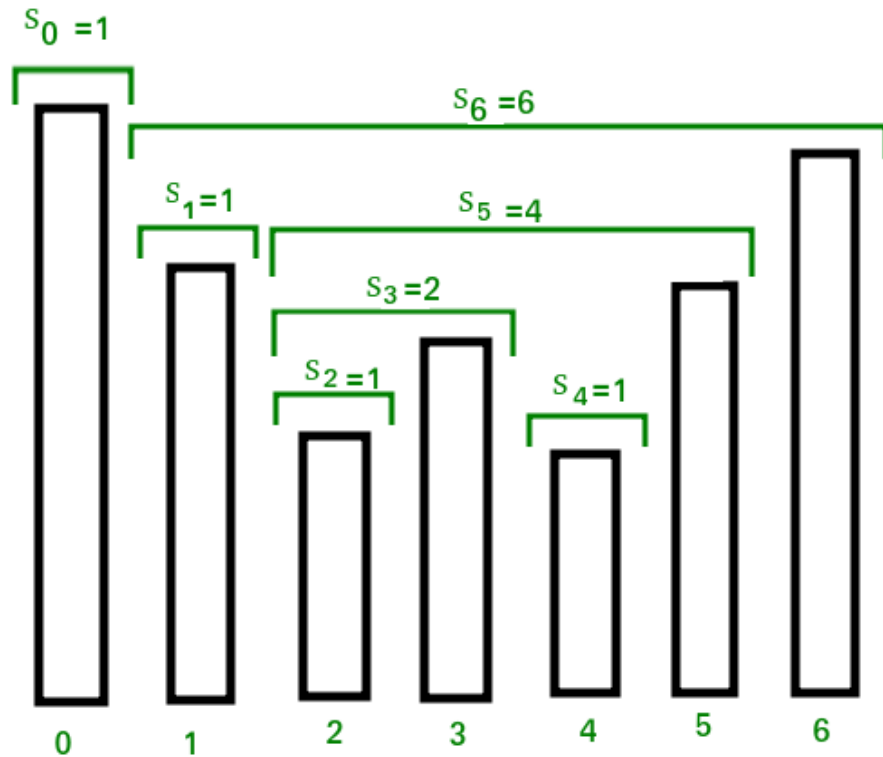
The Stock Span Problem

The Stock Span Problem - GeeksforGeeks

[The stock span problem](#) is a financial problem where we have a series of n daily price quotes for a stock and we need to calculate span of stock's price for all n days.

The span S_i of the stock's price on a given day i is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the current day is less than or equal to its price on the given day.

For example, if an array of 7 days prices is given as $\{100, 80, 60, 70, 60, 75, 85\}$, then the span values for corresponding 7 days are $\{1, 1, 1, 2, 1, 4, 6\}$



A Simple but inefficient method

Traverse the input price array. For every element being visited, traverse elements on left of it and increment the span value of it while elements on the left side are smaller.

Following is implementation of this method.

C

```
// C program for brute force method to calculate stock span values
#include <stdio.h>

// Fills array S[] with span values
void calculateSpan(int price[], int n, int S[])
{
    // Span value of first day is always 1
    S[0] = 1;

    // Calculate span value of remaining days by linearly checking
    // previous days
    for (int i = 1; i < n; i++)
    {
```



```
S[i] = 1; // Initialize span value

// Traverse left while the next element on left is smaller
// than price[i]
for (int j = i-1; (j>=0)&&(price[i]>=price[j]); j--)
    S[i]++;
}

// A utility function to print elements of array
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
}

// Driver program to test above function
int main()
{
    int price[] = {10, 4, 5, 90, 120, 80};
    int n = sizeof(price)/sizeof(price[0]);
    int S[n];

    // Fill the span values in array S[]
    calculateSpan(price, n, S);

    // print the calculated span values
    printArray(S, n);

    return 0;
}
```

Java

```
// Java implementation for brute force method to calculate stock span values

import java.util.Arrays;

class GFG
{
    // method to calculate stock span values
    static void calculateSpan(int price[], int n, int S[])
    {
        // Span value of first day is always 1
        S[0] = 1;

        // Calculate span value of remaining days by linearly checking
        // previous days
    }
}
```

```
    for (int i = 1; i < n; i++)
    {
        S[i] = 1; // Initialize span value

        // Traverse left while the next element on left is smaller
        // than price[i]
        for (int j = i-1; (j>=0)&&(price[i]>=price[j]); j--)
            S[i]++;
    }
}

// A utility function to print elements of array
static void printArray(int arr[])
{
    System.out.print(Arrays.toString(arr));
}

// Driver program to test above functions
public static void main(String[] args)
{
    int price[] = {10, 4, 5, 90, 120, 80};
    int n = price.length;
    int S[] = new int[n];

    // Fill the span values in array S[]
    calculateSpan(price, n, S);

    // print the calculated span values
    printArray(S);
}
}
// This code is contributed by Sumit Ghosh
```

Python

```
# Python program for brute force method to calculate stock span values

# Fills list S[] with span values
def calculateSpan(price, n, S):

    # Span value of first day is always 1
    S[0] = 1

    # Calculate span value of remaining days by linearly
    # checking previous days
    for i in range(1, n, 1):
        S[i] = 1 # Initialize span value
```

```
# Traverse left while the next element on left is
# smaller than price[i]
j = i - 1
while (j>=0) and (price[i] >= price[j]) :
    S[i] += 1
    j -= 1

# A utility function to print elements of array
def printArray(arr, n):

    for i in range(n):
        print(arr[i], end = " ")

# Driver program to test above function
price = [10, 4, 5, 90, 120, 80]
n = len(price)
S = [None] * n

# Fill the span values in list S[]
calculateSpan(price, n, S)

# print the calculated span values
printArray(S, n)

# This code is contributed by Sunny Karira
```

C#

```
// C# implementation for brute force method
// to calculate stock span values
using System;

class GFG {

    // method to calculate stock span values
    static void calculateSpan(int []price,
                              int n, int []S)
    {

        // Span value of first day is always 1
        S[0] = 1;

        // Calculate span value of remaining
        // days by linearly checking previous
        // days
        for (int i = 1; i < n; i++)
        {
```

```
        S[i] = 1; // Initialize span value

        // Traverse left while the next
        // element on left is smaller
        // than price[i]
        for (int j = i-1; (j >= 0) &&
            (price[i] >= price[j]); j--)
            S[i]++;
    }
}

// A utility function to print elements
// of array
static void printArray(int []arr)
{
    string result = string.Join(" ", arr);
    Console.WriteLine(result);
}

// Driver function
public static void Main()
{
    int []price = {10, 4, 5, 90, 120, 80};
    int n = price.Length;
    int []S= new int[n];

    // Fill the span values in array S[]
    calculateSpan(price, n, S);

    // print the calculated span values
    printArray(S);
}

}
```

// This code is contributed by Sam007.

PHP

```
<?php
// PHP program for brute force method
// to calculate stock span values

// Fills array S[] with span values
function calculateSpan($price, $n, $S)
{
    // Span value of first
```

```

// day is always 1
$S[0] = 1;

// Calculate span value of
// remaining days by linearly
// checking previous days
for ($i = 1; $i < $n; $i++)
{

    // Initialize span value
    $S[$i] = 1;

    // Traverse left while the next
    // element on left is smaller
    // than price[i]
    for ($j = $i - 1; ($j >= 0) &&
        ($price[$i] >= $price[$j]); $j--)
        $S[$i]++;
}

// print the calculated
// span values
for ($i = 0; $i < $n; $i++)
    echo $S[$i] . " ";

}

// Driver Code
$price = array(10, 4, 5, 90, 120, 80);
$n = count($price);
$S = array($n);

// Fill the span values in array S[]
calculateSpan($price, $n, $S);

// This code is contributed by Sam007
?>

```

Output :

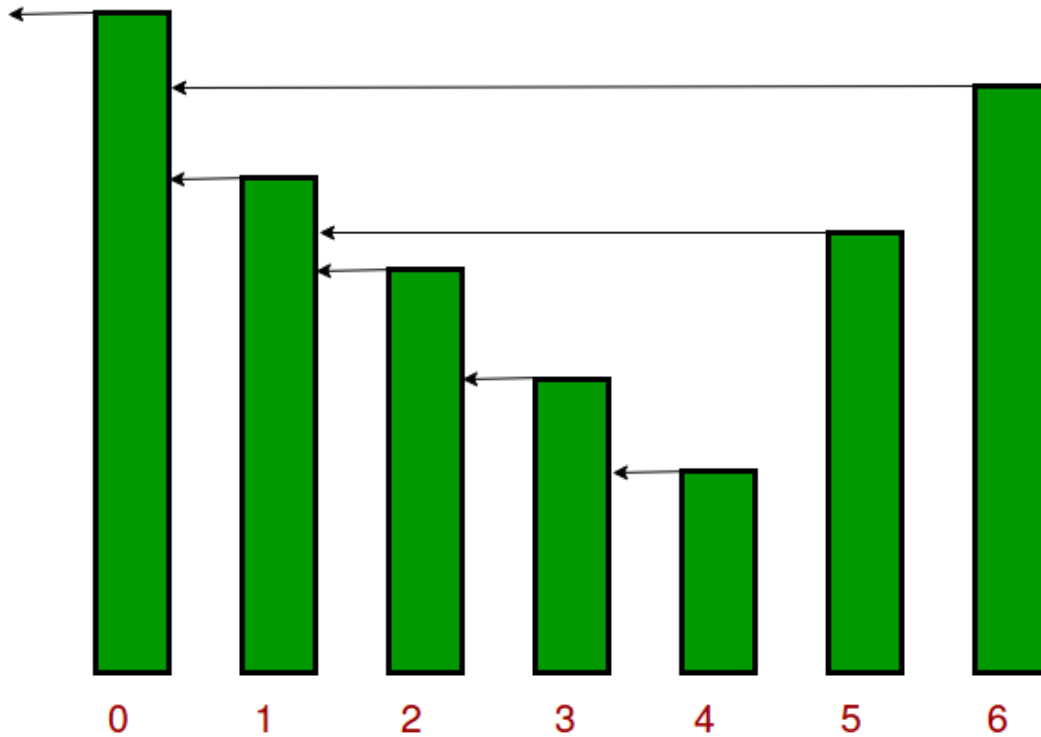
```
1 1 2 4 5 1
```

Time Complexity of the above method is $O(n^2)$. We can calculate stock span values in $O(n)$ time.

A Linear Time Complexity Method

We see that $S[i]$ on day i can be easily computed if we know the closest day preceding i , such that the price is greater than on that day than the price on day i . If such a day exists, let's call it $h(i)$, otherwise, we define $h(i) = -1$.

The span is now computed as $S[i] = i - h(i)$. See the following diagram.



To implement this logic, we use a stack as an abstract data type to store the days i , $h(i)$, $h(h(i))$ and so on. When we go from day $i-1$ to i , we pop the days when the price of the stock was less than or equal to $price[i]$ and then push the value of day i back into the stack.

Following is C++ implementation of this method.

C++

```
// a linear time solution for stock span problem
#include <iostream>
#include <stack>
using namespace std;

// A stack based efficient method to calculate stock span values
void calculateSpan(int price[], int n, int S[])
{
    // Create a stack and push index of first element to it
    stack<int> st;
```

```
st.push(0);

// Span value of first element is always 1
S[0] = 1;

// Calculate span values for rest of the elements
for (int i = 1; i < n; i++)
{
    // Pop elements from stack while stack is not empty and top of
    // stack is smaller than price[i]
    while (!st.empty() && price[st.top()] <= price[i])
        st.pop();

    // If stack becomes empty, then price[i] is greater than all elements
    // on left of it, i.e., price[0], price[1],..price[i-1]. Else price[i]
    // is greater than elements after top of stack
    S[i] = (st.empty())? (i + 1) : (i - st.top());

    // Push this element to stack
    st.push(i);
}

// A utility function to print elements of array
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}

// Driver program to test above function
int main()
{
    int price[] = {10, 4, 5, 90, 120, 80};
    int n = sizeof(price)/sizeof(price[0]);
    int S[n];

    // Fill the span values in array S[]
    calculateSpan(price, n, S);

    // print the calculated span values
    printArray(S, n);

    return 0;
}
```

Java

```
// Java linear time solution for stock span problem

import java.util.Stack;
import java.util.Arrays;

public class GFG
{
    // a linear time solution for stock span problem
    // A stack based efficient method to calculate stock span values
    static void calculateSpan(int price[], int n, int S[])
    {
        // Create a stack and push index of first element to it
        Stack<Integer> st= new Stack<>();
        st.push(0);

        // Span value of first element is always 1
        S[0] = 1;

        // Calculate span values for rest of the elements
        for (int i = 1; i < n; i++)
        {
            // Pop elements from stack while stack is not empty and top of
            // stack is smaller than price[i]
            while (!st.empty() && price[st.peek()] <= price[i])
                st.pop();

            // If stack becomes empty, then price[i] is greater than all elements
            // on left of it, i.e., price[0], price[1],..price[i-1]. Else price[i]
            // is greater than elements after top of stack
            S[i] = (st.empty())? (i + 1) : (i - st.peek());

            // Push this element to stack
            st.push(i);
        }
    }

    // A utility function to print elements of array
    static void printArray(int arr[])
    {
        System.out.print(Arrays.toString(arr));
    }

    // Driver method
    public static void main(String[] args)
    {
        int price[] = {10, 4, 5, 90, 120, 80};
        int n = price.length;
        int S[]=new int[n];
    }
}
```



```
        // Fill the span values in array S[]
        calculateSpan(price, n, S);

        // print the calculated span values
        printArray(S);
    }
}
// This code is contributed by Sumit Ghosh
```

Python

```
# A linear time solution for stack stock problem

# A stack based efficient method to calculate s
def calculateSpan(price, S):

    n = len(price)
    # Create a stack and push index of first element to it
    st = []
    st.append(0)

    # Span value of first element is always 1
    S[0] = 1

    # Calculate span values for rest of the elements
    for i in range(1, n):

        # Pop elements from stack while stack is not
        # empty and top of stack is smaller than price[i]
        while( len(st) > 0 and price[st[0]] <= price[i]):
            st.pop()

        # If stack becomes empty, then price[i] is greater
        # than all elements on left of it, i.e. price[0],
        # price[1], ..price[i-1]. Else the price[i] is
        # greater than elements after top of stack
        S[i] = i+1 if len(st) <= 0 else (i - st[0])

        # Push this element to stack
        st.append(i)

# A utility function to print elements of array
def printArray(arr, n):
    for i in range(0,n):
        print arr[i],
```

```
# Driver program to test above function
price = [10, 4, 5, 90, 120, 80]
S = [0 for i in range(len(price)+1)]

# Fill the span values in array S[]
calculateSpan(price, S)

# Print the calculated span values
printArray(S, len(price))

# This code is contributed by Nikhil Kumar Singh (nickzuck_007)
```

Output:

```
1 1 2 4 5 1
```

Time Complexity: $O(n)$. It seems more than $O(n)$ at first look. If we take a closer look, we can observe that every element of array is added and removed from stack at most once. So there are total $2n$ operations at most. Assuming that a stack operation takes $O(1)$ time, we can say that the time complexity is $O(n)$.

Auxiliary Space: $O(n)$ in worst case when all elements are sorted in decreasing order.

References:

[http://en.wikipedia.org/wiki/Stack_\(abstract_data_type\)#The_Stock_Span_Problem](http://en.wikipedia.org/wiki/Stack_(abstract_data_type)#The_Stock_Span_Problem)
<http://crypto.cs.mcgill.ca/~crepeau/CS250/2004/Stack-I.pdf>

Improved By : [Sam007](#)

Source

<https://www.geeksforgeeks.org/the-stock-span-problem/>

Chapter 120

Tracking current Maximum Element in a Stack

Tracking current Maximum Element in a Stack - GeeksforGeeks

Given a Stack, keep track of the maximum value in it. The maximum value may be the top element of the stack, but once a new element is pushed or an element is pop from the stack, the maximum element will be now from the rest of the elements.

Examples:

```
Input : 4 19 7 14 20
Output : Max Values in stack are
         4 19 19 19 20
```

```
Input : 40 19 7 14 20 5
Output : Max Values in stack are
         40 40 40 40 40 40
```

Method 1 (Brute-force): We keep pushing the elements in the main stack and whenever we are asked to return the maximum element, we traverse the stack and print the max element.

Time Complexity : $O(n)$

Auxiliary Space : $O(1)$

Method 2 (Efficient): An efficient approach would be to maintain an auxiliary stack while pushing element in the main stack. This auxiliary stack will keep track of the maximum element.

Below is the step by step algorithm to do this:

1. Create an auxiliary stack, say 'trackStack' to keep the track of maximum element
2. Push the first element to both mainStack and the trackStack.

3. Now from the second element, push the element to the main stack. Compare the element with the top element of the track stack, if the current element is greater than top of trackStack then push the current element to trackStack otherwise push the top element of trackStack again into it.
4. If we pop an element from the main stack, then pop an element from the trackStack as well.
5. Now to compute the maximum of the main stack at any point, we can simply print the top element of Track stack.

Step by step explanation :

Suppose the elements are pushed on to the stack in the order {4, 2, 14, 1, 18}

Step 1 : Push 4, Current max : 4

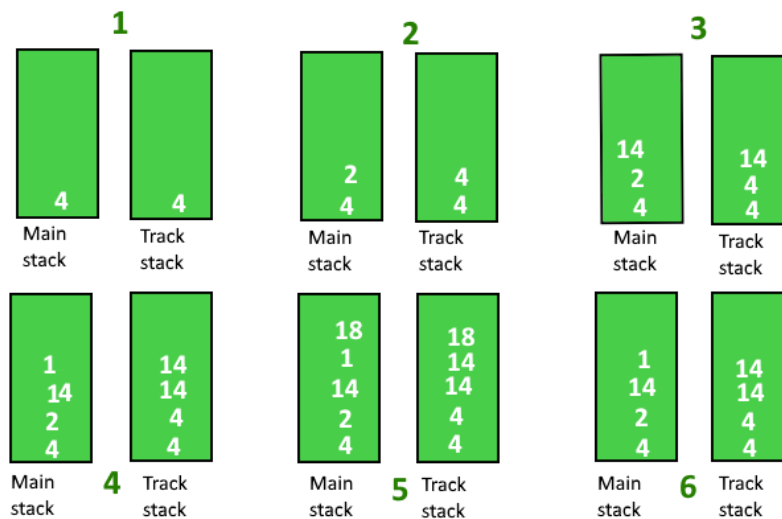
Step 2 : Push 2, Current max : 4

Step 3 : Push 14, Current max : 14

Step 4 : Push 1, Current max : 14

Step 5 : Push 18, Current max : 18

Step 6 : Pop 18, Current max : 14



Below is the C++ implementation of above approach:

```
// C++ program to keep track of maximum
// element in a stack
#include <bits/stdc++.h>
```

```
using namespace std;

class StackWithMax
{
    // main stack
    stack<int> mainStack;

    // tack to keep track of max element
    stack<int> trackStack;

public:
    void push(int x)
    {
        mainStack.push(x);
        if (mainStack.size() == 1)
        {
            trackStack.push(x);
            return;
        }

        // If current element is greater than
        // the top element of track stack, push
        // the current element to track stack
        // otherwise push the element at top of
        // track stack again into it.
        if (x > trackStack.top())
            trackStack.push(x);
        else
            trackStack.push(trackStack.top());
    }

    int getMax()
    {
        return trackStack.top();
    }

    int pop()
    {
        mainStack.pop();
        trackStack.pop();
    }
};

// Driver program to test above functions
int main()
{
    StackWithMax s;
    s.push(20);
```

```
    cout << s.getMax() << endl;
    s.push(10);
    cout << s.getMax() << endl;
    s.push(50);
    cout << s.getMax() << endl;
    return 0;
}
```

Output:

```
20
20
50
```

Time Complexity : $O(1)$

Auxiliary Complexity : $O(n)$

Source

<https://www.geeksforgeeks.org/tracking-current-maximum-element-in-a-stack/>

Chapter 121

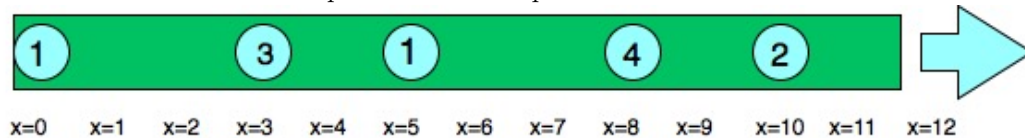
Water drop problem

Water drop problem - GeeksforGeeks

Consider a pipe of length L . The pipe has N water droplets at N different positions within it. Each water droplet is moving towards the end of the pipe ($x=L$) at different rates. When a water droplet mixes with another water droplet, it assumes the speed of the water droplet it is mixing with. Determine the no of droplets that come out of the end of the pipe.

Refer to the figure below:

Numbers on circles indicates speed of water droplets



Examples:

```
Input: length = 12, position = [10, 8, 0, 5, 3],  
       speed = [2, 4, 1, 1, 3]
```

Output: 3

Explanation:

Droplets starting at $x=10$ and $x=8$ become a droplet, meeting each other at $x=12$ at time = 1 sec.

The droplet starting at 0 doesn't mix with any other droplet, so it is a drop by itself.

Droplets starting at $x=5$ and $x=3$ become a single drop, mixing with each other at $x=6$ at time = 1 sec.

Note that no other droplets meet these drops before the end of the pipe, so the answer is 3.

Refer to the figure below

Numbers on circles indicates speed of water droplets.

Approach:

This problem uses greedy technique.

A drop will mix with another drop if two conditions are met:

1. If the drop is faster than the drop it is mixing with
2. If the position of the faster drop is behind the slower drop.

We use an array of **pairs** to store the position and the time that ith drop would take to reach the end of the pipe. Then we **sort** the array according to the position of the drops. Now we have a fair idea of which drops lie behind which drops and their respective time taken to reach the end. More time means less speed and less time means more speed. Now all the drops before a slower drop will mix with it. And all the drops after the slower drop will mix with the next slower drop and so on.

For example if the times to reach the end are- 12, 3, 7, 8, 1 (sorted according to positions) 0th drop is slowest, it won't mix with the next drop

1st drop is faster than the 2nd drop so they will mix and 2nd drop is faster than 3rd drop so all three will mix together. They cannot mix with the 4th drop because that is faster.

So we use a **stack** to maintain the local maxima of the times.

No of local maximal + residue(drops after last local maxima) = total no of drops

```
#include <bits/stdc++.h>
using namespace std;
int drops(int length, int position[], int speed[], int n)
{
    // stores position and time taken by a single
    // drop to reach the end as a pair
    vector<pair<int, double> > m(n);

    int i;
    for (i = 0; i < n; i++) {

        // calculates distance needs to be
        // covered by the ith drop
        int p = length - position[i];

        // inserts initial position of the
        // ith drop to the pair
        m[i].first = position[i];

        // inserts time taken by ith drop to reach
        // the end to the pair
        m[i].second = p * 1.0 / speed[i];
    }

    // sorts the pair according to increasing
    // order of their positions
    sort(m.begin(), m.end());
    int k = 0; // counter for no of final drops

    // stack to maintain the next slower drop
```



```
// which might coalesce with the current drop
stack<double> s;

// we traverse the array demo right to left
// to determine the slower drop
for (i = n - 1; i >= 0; i--)
{
    if (s.empty()) {
        s.push(m[i].second);
    }

    // checks for next slower drop
    if (m[i].second > s.top())
    {
        s.pop();
        k++;
        s.push(m[i].second);
    }
}

// calculating residual drops in the pipe
if (!s.empty())
{
    s.pop();
    k++;
}
return k;
}

// driver function
int main()
{
    int length = 12; // length of pipe
    int position[] = { 10, 8, 0, 5, 3 }; // position of droplets
    int speed[] = { 2, 4, 1, 1, 3 }; // speed of each droplets
    int n = sizeof(speed)/sizeof(speed[0]);
    cout << drops(length, position, speed, n);
    return 0;
}
```

Output:

3

Source

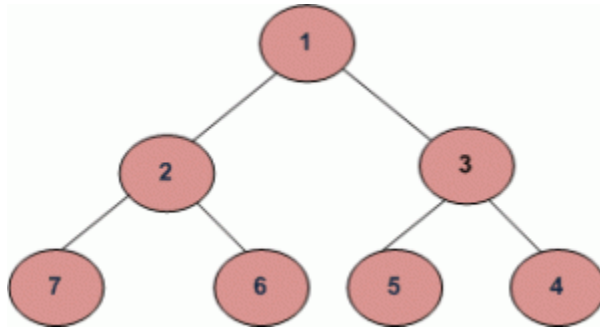
<https://www.geeksforgeeks.org/water-drop-problem/>

Chapter 122

ZigZag Tree Traversal

ZigZag Tree Traversal - GeeksforGeeks

Write a function to print ZigZag order traversal of a binary tree. For the below binary tree the zigzag order traversal will be **1 3 2 7 6 5 4**



This problem can be solved using two stacks. Assume the two stacks are current: **currentlevel** and **nextlevel**. We would also need a variable to keep track of the current level order(whether it is left to right or right to left). We pop from the currentlevel stack and print the nodes value. Whenever the current level order is from left to right, push the nodes left child, then its right child to the stack nextlevel. Since a stack is a LIFO(Last-In-First_out) structure, next time when nodes are popped off nextlevel, it will be in the reverse order. On the other hand, when the current level order is from right to left, we would push the nodes right child first, then its left child. Finally, do-not forget to swap those two stacks at the end of each level(i.e., when current level is empty)

Below is the implementation of the above approach:

C++

```
// C++ implementation of a O(n) time method for
// Zigzag order traversal
#include <iostream>
```

```

#include <stack>
using namespace std;

// Binary Tree node
struct Node {
    int data;
    struct Node *left, *right;
};

// function to print the zigzag traversal
void zigzagtraversal(struct Node* root)
{
    // if null then return
    if (!root)
        return;

    // declare two stacks
    stack<struct Node*> currentlevel;
    stack<struct Node*> nextlevel;

    // push the root
    currentlevel.push(root);

    // check if stack is empty
    bool lefttoright = true;
    while (!currentlevel.empty()) {

        // pop out of stack
        struct Node* temp = currentlevel.top();
        currentlevel.pop();

        // if not null
        if (temp) {

            // print the data in it
            cout << temp->data << " ";

            // store data according to current
            // order.
            if (lefttoright) {
                if (temp->left)
                    nextlevel.push(temp->left);
                if (temp->right)
                    nextlevel.push(temp->right);
            }
            else {
                if (temp->right)
                    nextlevel.push(temp->right);
            }
        }
    }
}

```

```

        if (temp->left)
            nextlevel.push(temp->left);
    }
}

    if (currentlevel.empty()) {
        lefttoright = !lefttoright;
        swap(currentlevel, nextlevel);
    }
}
}

// A utility function to create a new node
struct Node* newNode(int data)
{
    struct Node* node = new struct Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// driver program to test the above function
int main()
{
    // create tree
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(7);
    root->left->right = newNode(6);
    root->right->left = newNode(5);
    root->right->right = newNode(4);
    cout << "ZigZag Order traversal of binary tree is \n";

    zigzagtraversal(root);

    return 0;
}

```

Java

```

// Java implementation of a O(n) time
// method for Zigzag order traversal
import java.util.*;

// Binary Tree node
class Node
{

```

```
int data;
Node leftChild;
Node rightChild;
Node(int data)
{
    this.data = data;
}
}

class BinaryTree {
Node rootNode;

// function to print the
// zigzag traversal
void printZigZagTraversal() {

    // if null then return
    if (rootNode == null) {
        return;
    }

    // declare two stacks
    Stack<Node> currentLevel = new Stack<>();
    Stack<Node> nextLevel = new Stack<>();

    // push the root
    currentLevel.push(rootNode);
    boolean leftToRight = true;

    // check if stack is empty
    while (!currentLevel.isEmpty()) {

        // pop out of stack
        Node node = currentLevel.pop();

        // print the data in it
        System.out.print(node.data + " ");

        // store data according to current
        // order.
        if (leftToRight) {
            if (node.leftChild != null) {
                nextLevel.push(node.leftChild);
            }

            if (node.rightChild != null) {
                nextLevel.push(node.rightChild);
            }
        }
    }
}
```

```

    }
    else {
        if (node.rightChild != null) {
            nextLevel.push(node.rightChild);
        }

        if (node.leftChild != null) {
            nextLevel.push(node.leftChild);
        }
    }

    if (currentLevel.isEmpty()) {
        leftToRight = !leftToRight;
        Stack<Node> temp = currentLevel;
        currentLevel = nextLevel;
        nextLevel = temp;
    }
}
}

public class zigZagTreeTraversal {

    // driver program to test the above function
    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree();
        tree.rootNode = new Node(1);
        tree.rootNode.leftChild = new Node(2);
        tree.rootNode.rightChild = new Node(3);
        tree.rootNode.leftChild.leftChild = new Node(7);
        tree.rootNode.leftChild.rightChild = new Node(6);
        tree.rootNode.rightChild.leftChild = new Node(5);
        tree.rootNode.rightChild.rightChild = new Node(4);

        System.out.println("ZigZag Order traversal of binary tree is");
        tree.printZigZagTraversal();
    }
}

// This Code is contributed by Harikrishnan Rajan.

```

Python3

```

# Python Program to print zigzag traversal
# of binary tree

# Binary tree node

```

```
class Node:
    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = self.right = None

# function to print zigzag traversal of
# binary tree
def zigzagtraversal(root):

    # Base Case
    if root is None:
        return

    # Create two stacks to store current
    # and next level
    currentLevel = []
    nextLevel = []

    # if ltr is true push nodes from
    # left to right otherwise from
    # right to left
    ltr = True

    # append root to currentlevel stack
    currentLevel.append(root)

    # Check if stack is empty
    while len(currentLevel) > 0:
        # pop from stack
        temp = currentLevel.pop(-1)
        # print the data
        print(temp.data, " ", end="")

        if ltr:
            # if ltr is true push left
            # before right
            if temp.left:
                nextLevel.append(temp.left)
            if temp.right:
                nextLevel.append(temp.right)
        else:
            # else push right before left
            if temp.right:
                nextLevel.append(temp.right)
            if temp.left:
                nextLevel.append(temp.left)
```

```
        if len(currentLevel) == 0:
            # reverse ltr to push node in
            # opposite order
            ltr = not ltr
            # swapping of stacks
            currentLevel, nextLevel = nextLevel, currentLevel

# Driver program to check above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(7)
root.left.right = Node(6)
root.right.left = Node(5)
root.right.right = Node(4)
print("Zigzag Order traversal of binary tree is")
zigzagtraversal(root)

# This code is contributed by Shweta Singh
```

Output:

```
ZigZag Order traversal of binary tree is
1 3 2 7 6 5 4
```

Time Complexity: $O(n)$ **Space Complexity:** $O(n) + (n) = O(n)$ **Improved By :** [shweta44](#)**Source**<https://www.geeksforgeeks.org/zigzag-tree-traversal/>