# Copyright

The content of this book comes from [geeksforgeeks.org](http://geeksforgeeks.org) and it's licensed under [Creative Commons Attribution-NonCommercial-NoDerivs 2.5 India](http://creativecommons.org/licenses/by-nc-nd/2.5/in/deed.en_US)

Made by [Jing](http://www.jing-zhou.me/). 2015.

Updated on August 13, 2015

Head over to [this github repo](https://github.com/gnijuohz/geeksforgeeks-as-books) to report issues or contribute.

# Design a stack with operations on middle element

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 C 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.

/\* 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;  
}

Output:

Item popped is 77  
Item popped is 66  
Middle Element is 33

This article is contributed by [**Chandra Prakash**](https://www.facebook.com/chandra.prakash.52643). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

### Source

<http://www.geeksforgeeks.org/design-a-stack-with-find-middle-operation/>

Category: [Linked Lists](http://www.geeksforgeeks.org/category/linked-list/) Tags: [stack](http://www.geeksforgeeks.org/tag/stack/)

# Design and Implement Special Stack Data Structure | Added Space Optimized Version

**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 C++ implementation for SpecialStack class. In the below implementation, SpecialStack inherits from Stack and has one Stack object min which work as auxiliary stack.  
   
#include<iostream>  
#include<stdlib.h>  
  
using namespace std;  
  
/\* A simple stack class with basic stack funtionalities \*/  
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  
 auxiliarry 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;  
}

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().

/\* 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(x);  
  
 return x;  
}

Thanks to [@Venki](http://www.geeksforgeeks.org/archives/14149/comment-page-1#comment-5366), [@swarup](http://www.geeksforgeeks.org/archives/14149/comment-page-1#comment-5359) and [@Jing Huang](http://www.geeksforgeeks.org/archives/14149/comment-page-1#comment-5369) for their inputs.

Please write comments if you find the above code incorrect, or find other ways to solve the same problem.

### Source

<http://www.geeksforgeeks.org/design-and-implement-special-stack-data-structure/>

# How to efficiently implement k stacks in a single array?

We have discussed [space efficient implementation of 2 stacks in a single array](http://www.geeksforgeeks.org/implement-two-stacks-in-an-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 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 C++ implementation of the above idea.

// 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;  
}

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

<http://www.geeksforgeeks.org/efficiently-implement-k-stacks-single-array/>

# Find maximum depth of nested parenthesis in a string

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.

More examples:

S = "( a(b) (c) (d(e(f)g)h) I (j(k)l)m)";  
Output : 4  
  
S = "( p((q)) ((s)t) )";  
Output : 3  
  
S = "";  
Output : 0  
  
S = "b) (c) ()";  
Output : -1   
  
S = "(b) ((c) ()"  
Output : -1

Source : Walmart Labs Interview Question

**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 C++ implementation of above algorithm.

// 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;  
}

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.

### Source

<http://www.geeksforgeeks.org/find-maximum-depth-nested-parenthesis-string/>

Category: [Strings](http://www.geeksforgeeks.org/category/c-strings/) Tags: [stack](http://www.geeksforgeeks.org/tag/stack/), [Stack-Queue](http://www.geeksforgeeks.org/tag/stack-queue/), [StackAndQueue](http://www.geeksforgeeks.org/tag/stackandqueue/)

# Implement Stack using Queues

The problem is opposite of [this](http://www.geeksforgeeks.org/archives/5009)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.

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.

**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.

**References:**  
 [Implement Stack using Two Queues](http://stackoverflow.com/questions/688276/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

<http://www.geeksforgeeks.org/implement-stack-using-queue/>

Category: [Misc](http://www.geeksforgeeks.org/category/c-programs/) Tags: [Queue](http://www.geeksforgeeks.org/tag/queue/), [stack](http://www.geeksforgeeks.org/tag/stack/)

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.

#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;  
}

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.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

### Source

<http://www.geeksforgeeks.org/implement-two-stacks-in-an-array/>

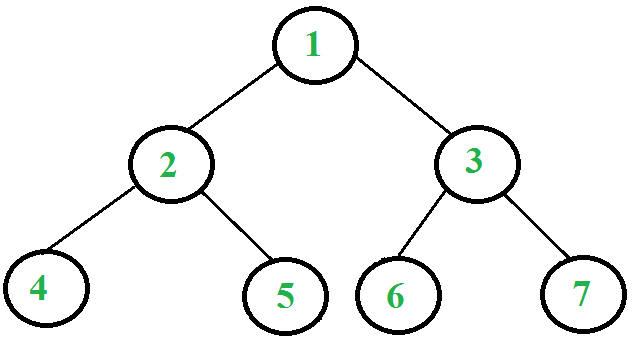
# Iterative Postorder Traversal | Set 2 (Using One Stack)

We have discussed a simple [iterative postorder traversal using two stacks](http://www.geeksforgeeks.org/iterative-postorder-traversal/) 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  
 [](http://d2dskowxfbo68o.cloudfront.net/wp-content/uploads/BinaryTree.png)

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 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);  
  
 postOrderIterative(root);  
  
 return 0;  
}

Output:

4 5 2 6 7 3 1

This article is compiled by[Aashish Barnwal](https://www.facebook.com/barnwal.aashish?fref=ts). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

### Source

<http://www.geeksforgeeks.org/iterative-postorder-traversal-using-stack/>

Category: [Trees](http://www.geeksforgeeks.org/category/tree/) Tags: [stack](http://www.geeksforgeeks.org/tag/stack/)

Post navigation

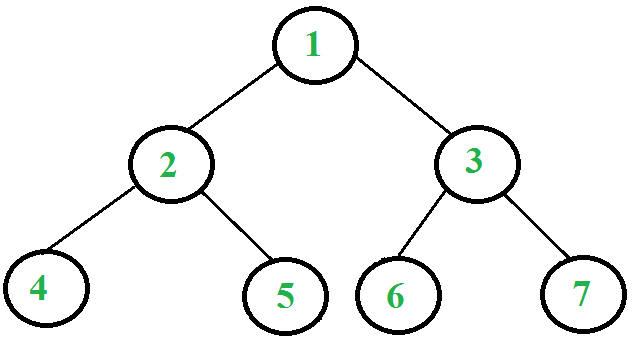
[← Iterative Postorder Traversal | Set 1 (Using Two Stacks)](http://www.geeksforgeeks.org/iterative-postorder-traversal/) [Generate n-bit Gray Codes →](http://www.geeksforgeeks.org/given-a-number-n-generate-bit-patterns-from-0-to-2n-1-so-that-successive-patterns-differ-by-one-bit/)

# Iterative Postorder Traversal | Set 1 (Using Two Stacks)

We have discussed [iterative inorder](http://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/) and [iterative preorder](http://www.geeksforgeeks.org/iterative-preorder-traversal/) 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](http://en.wikipedia.org/wiki/Tail_call), there is an extra statement after the final recursive call to itself). The postorder traversal can easily be done using two stacks though. The idea is to push reverse postorder traversal to a stack. Once we have reverse 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 LIFO property of stacks. Now the question is, how to get reverse post order elements in a stack – the other 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 preorder traversal. The only difference is right child is visited before left child and therefore sequence is “root right left” instead of “root left right”. So we can do something like [iterative preorder traversal](http://www.geeksforgeeks.org/iterative-preorder-traversal/) with following differences.  
 a) Instead of printing an item, we push it to a stack.  
 b) We push left subtree before right subtree.

Following is the complete algorithm. After step 2, we get reverse postorder traversal in second stack. We use first stack to get this 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  
 [](http://d2dskowxfbo68o.cloudfront.net/wp-content/uploads/BinaryTree.png)

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 since there is no more item in first stack.   
Observe that content of second stack is in postorder fashion. Print them.

Following is C implementation of iterative postorder traversal using two stacks.

#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)  
{  
 // 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;  
}

Output:

4 5 2 6 7 3 1

Following is 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 in second stack. The steps to get reverse postorder are similar to [iterative preorder](http://www.geeksforgeeks.org/iterative-preorder-traversal/).

You may also like to see [a method which uses only one stack](http://www.geeksforgeeks.org/iterative-postorder-traversal-using-stack/).

This article is compiled by[Aashish Barnwal](https://www.facebook.com/barnwal.aashish?fref=ts). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

### Source

<http://www.geeksforgeeks.org/iterative-postorder-traversal/>

Category: [Trees](http://www.geeksforgeeks.org/category/tree/) Tags: [stack](http://www.geeksforgeeks.org/tag/stack/)

Post navigation

[← Flatten a multilevel linked list](http://www.geeksforgeeks.org/flatten-a-linked-list-with-next-and-child-pointers/) [Iterative Postorder Traversal | Set 2 (Using One Stack) →](http://www.geeksforgeeks.org/iterative-postorder-traversal-using-stack/)

# Iterative Tower of Hanoi

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](http://geeksquiz.com/c-program-for-tower-of-hanoi/). We have also seen that, for n disks, total 2n – 1 moves are required.

**Iterative Algorithm:**

1. Calculate the total number of moves required i.e. "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 % 4 == 1) legal movement between ‘S’ and ‘D’   
  
  
  
When i = 5, (i % 5 == 2) legal movement between ‘S’ and ‘A’   
  
  
When i = 6, (i % 6 == 0) legal movement between ‘A’ and ‘D’   
  
  
When i = 7, (i % 7 == 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 thatafter 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 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;  
}

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'

**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

### Source

<http://www.geeksforgeeks.org/iterative-tower-of-hanoi/>

Category: [Misc](http://www.geeksforgeeks.org/category/c-programs/) Tags: [stack](http://www.geeksforgeeks.org/tag/stack/), [Stack-Queue](http://www.geeksforgeeks.org/tag/stack-queue/), [StackAndQueue](http://www.geeksforgeeks.org/tag/stackandqueue/)

Post navigation

[← VMWare Interview Experience | Set 3](http://www.geeksforgeeks.org/vmware-interview-experience-set-3/) [Interview Experience with Yatra(Hyderabad) | Set 3 (For Senior Software Engineer) →](http://www.geeksforgeeks.org/interview-experience-with-yatrahyderabad-set-3-for-senior-software-engineer/)

# Largest Rectangular Area in a Histogram | Set 2

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)

[](http://d2dskowxfbo68o.cloudfront.net/wp-content/uploads/histogram1.png)

We have discussed a [Divide and Conquer based O(nLogn) solution](http://www.geeksforgeeks.org/largest-rectangular-area-in-a-histogram-set-1/) for this problem. In this post, O(n) time solution is discussed. Like the [previous post](http://www.geeksforgeeks.org/largest-rectangular-area-in-a-histogram-set-1/), 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 C++ implementation of the above algorithm.

// 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; // Initalize 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;  
}

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](https://www.facebook.com/aasshishh?fref=ts) 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

<http://www.geeksforgeeks.org/largest-rectangle-under-histogram/>

# Merge Overlapping Intervals

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 interval[i] doesn’t overlap with interval[i-1], then interval[i+1] cannot overlap with interval[i-1] because starting time of interval[i+1] must be greater than or equal to 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 <iostream>  
#include <vector>  
#include <algorithm>  
#include <stack>  
using namespace std;  
  
// An interval has start time and end time  
struct Interval  
{  
 int start;  
 int end;  
};  
  
// Compares two intervals according to their staring 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)? true: false; }  
  
// The main function that takes a set of intervals, merges  
// overlapping intervals and prints the result  
void mergeIntervals(vector<Interval>& intervals)  
{  
 // Test if the given set has at least one interval  
 if (intervals.size() <= 0)  
 return;  
  
 // Create an empty stack of intervals  
 stack<Interval> s;  
  
 // sort the intervals based on start time  
 sort(intervals.begin(), intervals.end(), compareInterval);  
  
 // push the first interval to stack  
 s.push(intervals[0]);  
  
 // Start from the next interval and merge if necessary  
 for (int i = 1 ; i < intervals.size(); 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 < intervals[i].start)  
 {  
 s.push( intervals[i] );  
 }  
 // Otherwise update the ending time of top if ending of current   
 // interval is more  
 else if (top.end < intervals[i].end)  
 {  
 top.end = intervals[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;  
}  
  
// Functions to run test cases  
void TestCase1()  
{  
 // Create a set of intervals  
 Interval intvls[] = { {6,8}, {1,9}, {2,4}, {4,7} };  
 vector<Interval> intervals(intvls, intvls+4);  
  
 // Merge overlapping inervals and print result  
 mergeIntervals(intervals);  
}  
void TestCase2()  
{  
 // Create a set of intervals  
 Interval intvls[] = { {6,8},{1,3},{2,4},{4,7} };  
 vector<Interval> intervals(intvls, intvls+4);  
  
 // Merge overlapping inervals and print result  
 mergeIntervals(intervals);  
}  
void TestCase3()  
{  
 // Create a set of intervals  
 Interval intvls[] = { {1,3},{7,9},{4,6},{10,13} };  
 vector<Interval> intervals(intvls, intvls+4);  
  
 // Merge overlapping inervals and print result  
 mergeIntervals(intervals);  
}  
  
// Driver program to test above functions  
int main()  
{  
 TestCase1();  
 TestCase2();  
 TestCase3();  
 return 0;  
}

Output:

The Merged Intervals are: [1,9]  
 The Merged Intervals are: [1,8]  
 The Merged Intervals are: [10,13] [7,9] [4,6] [1,3]

Time complexity of the method is O(nLogn) which is for sorting. Once the array of intervals is sorted, merging takes linear time.

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

<http://www.geeksforgeeks.org/merging-intervals/>

Category: [Arrays](http://www.geeksforgeeks.org/category/c-arrays/) Tags: [stack](http://www.geeksforgeeks.org/tag/stack/)

Post navigation

[← Construct Complete Binary Tree from its Linked List Representation](http://www.geeksforgeeks.org/given-linked-list-representation-of-complete-tree-convert-it-to-linked-representation/) [Amazon Interview | Set 23 →](http://www.geeksforgeeks.org/amazon-interview-set-23/)

# Print ancestors of a given binary tree node without recursion

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

1  
 / \  
 2 3  
 / \ / \  
 4 5 6 7   
 / \ /  
 8 9 10

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](http://www.geeksforgeeks.org/print-ancestors-of-a-given-node-in-binary-tree/).  
 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 C implementation of the above approach.

// 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;  
}

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.

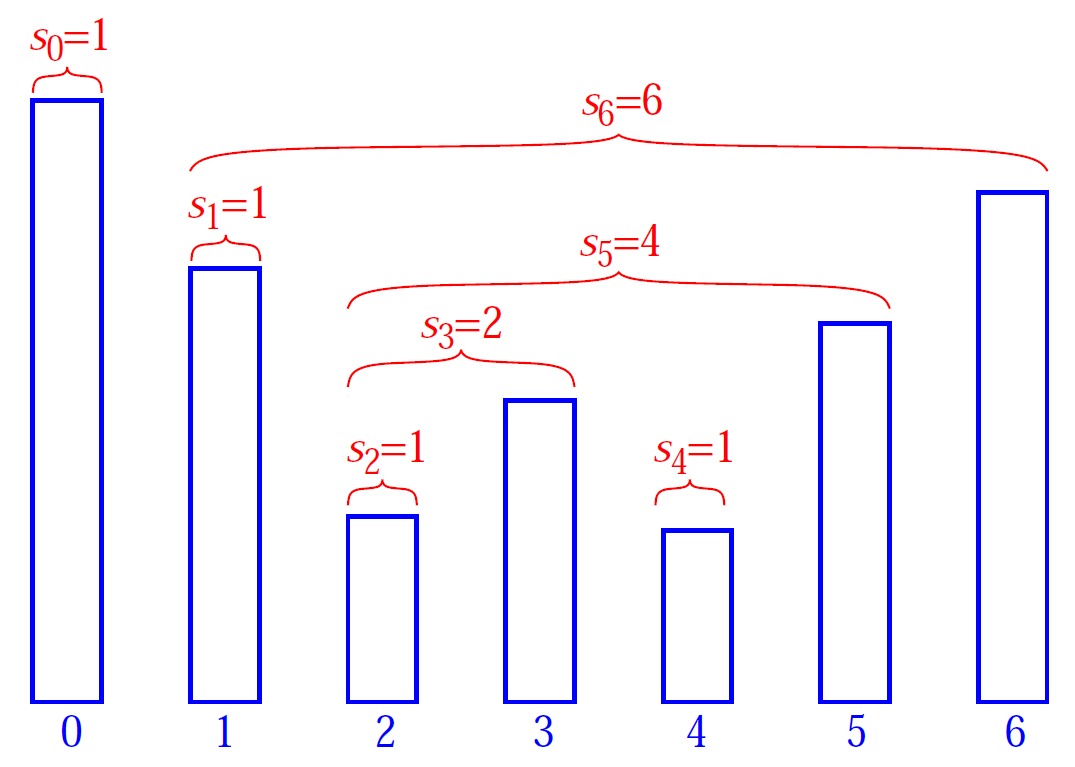
This article is contrubuted by[**Chandra Prakash**](https://www.facebook.com/chandra.prakash.52643?fref=ts). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

### Source

<http://www.geeksforgeeks.org/print-ancestors-of-a-given-binary-tree-node-without-recursion/>

Category: [Trees](http://www.geeksforgeeks.org/category/tree/) Tags: [stack](http://www.geeksforgeeks.org/tag/stack/), [StackAndQueue](http://www.geeksforgeeks.org/tag/stackandqueue/)

# The Stock Span Problem

[The stock span problem](http://en.wikipedia.org/wiki/Stack_(abstract_data_type)#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 Si 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}  
 [](http://d2dskowxfbo68o.cloudfront.net/wp-content/uploads/StockSpanProblem.png)

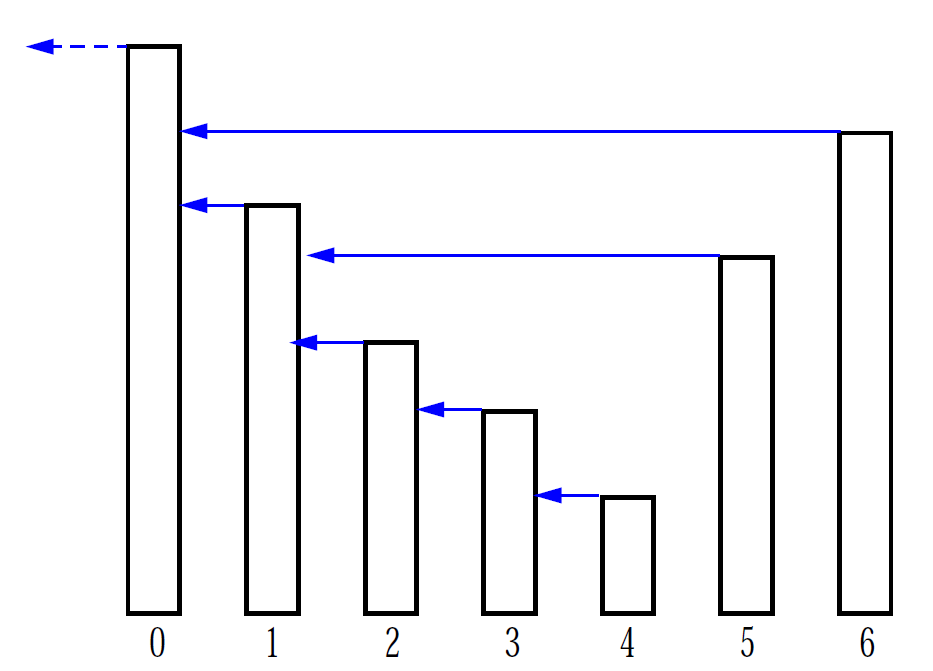
**A Simple but inefficient method**  
 Traverse the input price array. For every element being visited, traverse elements on left of it and increment the span value of it while elements on the left side are smaller.

Following is implementation of this method.

// A 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;  
}

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.

[](http://d2dskowxfbo68o.cloudfront.net/wp-content/uploads/StockSpanProblem2.png)

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.

// a linear time solution for stock span problem  
#include <iostream>  
#include <stack>  
using namespace std;  
  
// A brute force 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;  
}

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://crypto.cs.mcgill.ca/~crepeau/CS250/2004/Stack-I.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

### Source

<http://www.geeksforgeeks.org/the-stock-span-problem/>