Contents

Chapter 1

Implement two stacks in an array

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;
}
```

Python

```
class twoStacks:
    arr = [None]
    size,top1,top2 = 0,0,0

    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()

    # 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()))
```

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/

Chapter 2

Check for balanced parentheses in an expression

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 = “[(])”

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”

Implementation:

```
#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 a 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("\n Balanced ");
  else
    printf("\n Not Balanced ");  \
  getchar();
}

/* 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;
  }
}
```

Time Complexity: O(n)
Auxiliary Space: O(n) for stack.

Please write comments if you find any bug in above codes/algorithms, or find other ways to solve the same problem

Source

http://www.geeksforgeeks.org/check-for-balanced-parentheses-in-an-expression/

Category: Misc

Chapter 3

Next Greater Element

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 Sachinfor providing following code.

C/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 -- %d\n", arr[i], next);
    }
}

int main()
{
    int arr[]= {11, 13, 21, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    printNGE(arr, n);
    getchar();
    return 0;
}
```

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
```

Output:

```
11 -- 13
13 -- 21
21 -- -1
3 -- -1
```

Time Complexity: O(nˆ2). The worst case occurs when all elements are sorted in decreasing order.

Method 2 (Using Stack)
Thanks to pchildfor suggesting following approach.
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
....g) If *next* is smaller than the popped element, then push the popped element back.
3) After the loop in step 2 is over, pop all the elements from stack and print -1 as next element for them.

C

```c
// A Stack based C program to find next greater element
// for all array elements.
#include<stdio.h>
#include<stdlib.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 overflow\n");
        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;
}
```

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.

Source:

http://geeksforgeeks.org/forum/topic/next-greater-element#post-60

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

Category: Arrays

Post navigation

← Reverse alternate K nodes in a Singly Linked List Practice Questions for Recursion | Set 5 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Source

http://www.geeksforgeeks.org/next-greater-element/

Chapter 4

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)

Solution:
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

First 4 is inserted at the bottom.
    4
So we need a function that inserts at the bottom of a stack using the above given basic stack function. //Belo

void insertAtBottom(struct sNode** top_ref, int item)
{
   int temp;
   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 */
     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)
{
  int temp;
  if(!isEmpty(*top_ref))
  {

    /* Hold all items in Function Call Stack until we reach end of
     the stack */
    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);
  }
}
```

//Below is a complete running program for testing above functions.

```
#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);

/* Driveer program to test above functions */
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);
  getchar();
}

/* 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");
     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;
  }
}

/* Functrion to pront a linked list */
void print(struct sNode* top)
{
  printf("\n");
  while(top != NULL)
  {
    printf(" %d ", top->data);
    top =  top->next;
  }
}
```

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/reverse-a-stack-using-recursion/

Category: Misc Tags: Recursion

Chapter 5

Sort a stack using recursion

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
We strongly recommend you to minimize your browser and try this yourself first.

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. When the s


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 is 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 poped element in variable 'temp'.  After poping all the el

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
```

Next 18 (from stack frame #3) is picked. Since 18 30 **18** -5

Next 14 (from stack frame #2) is picked. Since 14 30 **14** -5

Now -3 (from stack frame #1) is picked, as -3 30 -3 -5

Implementation:
Below is C implementation of above algorithm.

```
// 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 chcek 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;
}
```

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

http://www.geeksforgeeks.org/sort-a-stack-using-recursion/

Category: Misc Tags: Recursion, stack

Post navigation

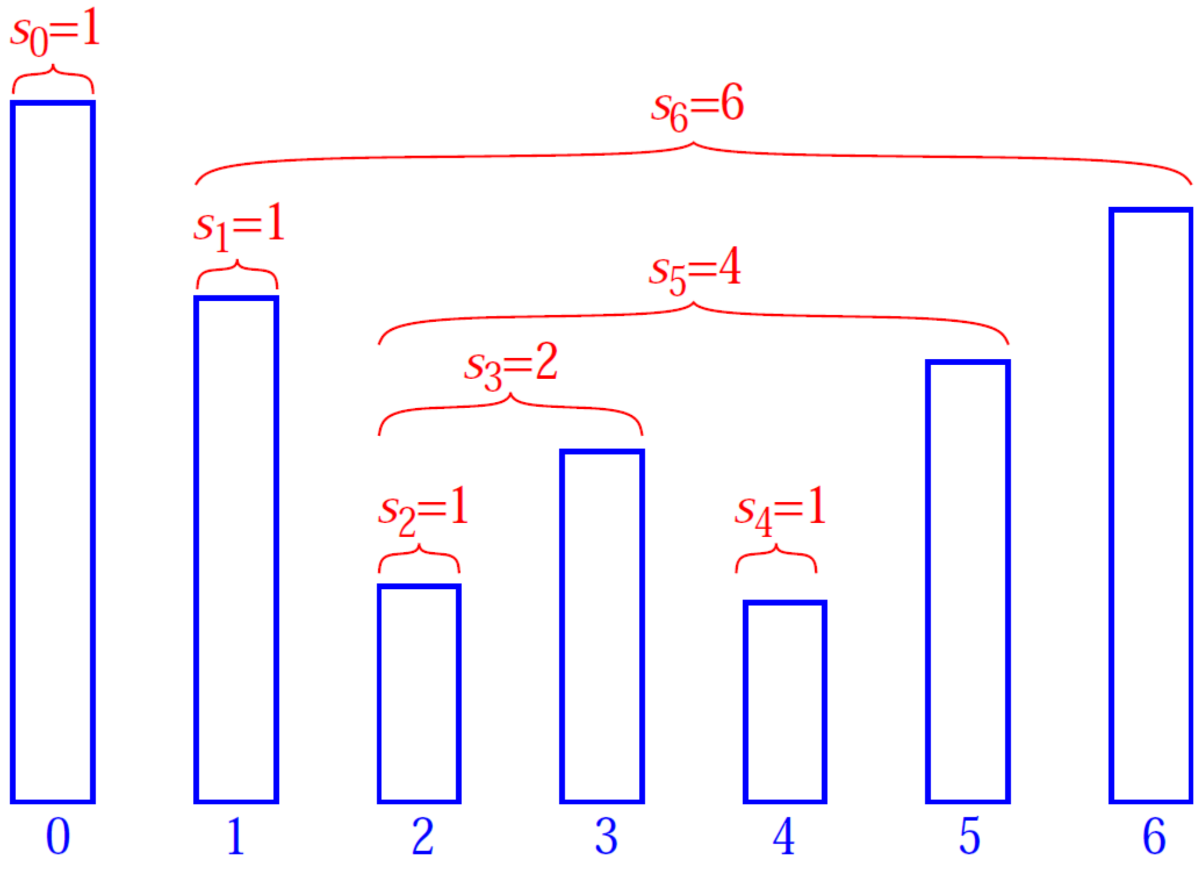
← Hopcroft–Karp Algorithm for Maximum Matching | Set 2 (Implementation) Find the longest path in a matrix with given constraints →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 6

The Stock Span Problem

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}



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.

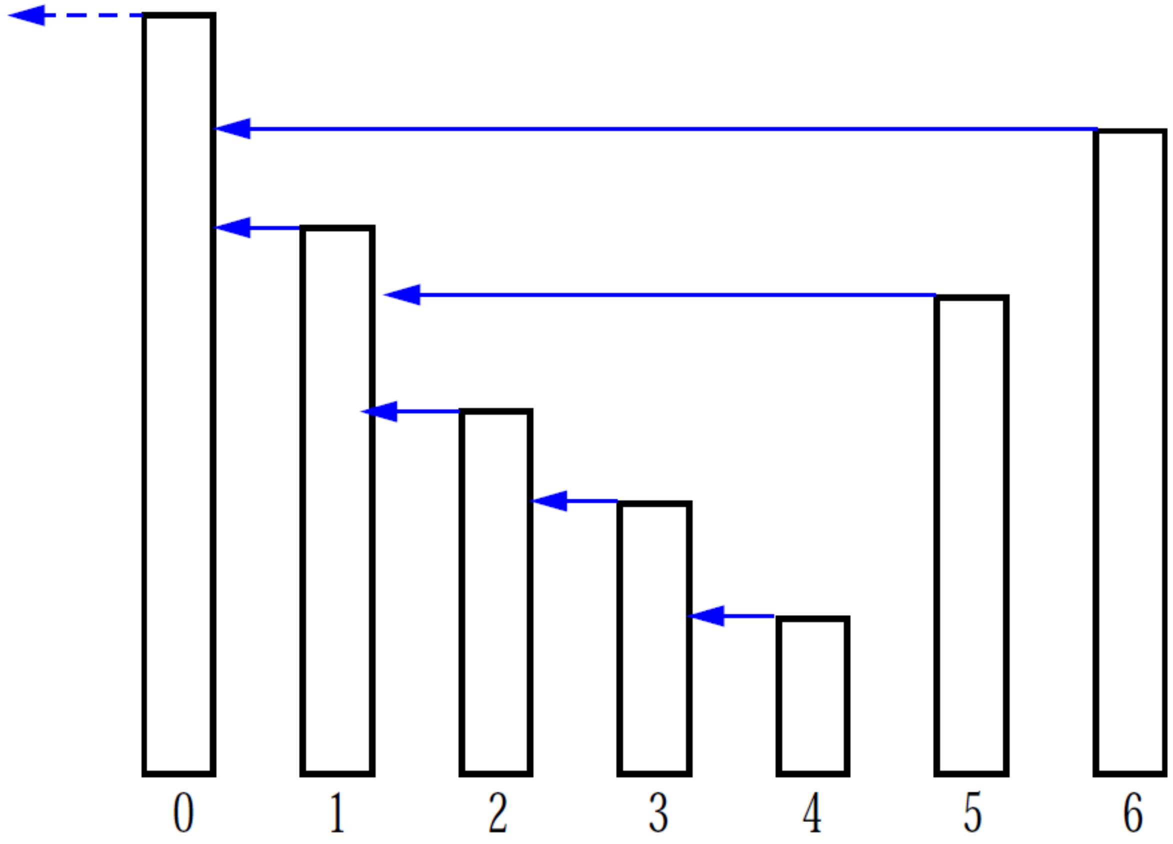
Following is implementation of this method.

```
// 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.



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 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;
}
```

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/

Chapter 7

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 f

#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(y);

    return x;
}
```

Thanks to @Venki, @swarup and @Jing Huang 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/

Chapter 8

Implement Stack using Queues

The problem is opposite of thispost. 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

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 Tags: Queue, stack

Chapter 9

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**. 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 Tags: stack

Chapter 10

How to efficiently implement k stacks in a single array?

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 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/

Chapter 11

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. 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. "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 Tags: stack, Stack-Queue, StackAndQueue

Post navigation

← VMWare Interview Experience | Set 3 Interview Experience with Yatra(Hyderabad) | Set 3 (For Senior Software Engineer) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 12

Length of the longest valid substring

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:  ()(()))
```

We strongly recommend you to minimize your browser and try this yourself first.

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 is C++ implementation of above algorithm.

```
// 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;
}
```

Output:

```
 4
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.

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/length-of-the-longest-valid-substring/

Category: Misc Tags: stack, Stack-Queue

Post navigation

← Worms ,Viruses and beyond !! Hopcroft–Karp Algorithm for Maximum Matching | Set 1 (Introduction) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 13

Find maximum of minimum for every window size in a given array

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.
```

We strongly recommend you to minimize your browser and try this yourself first.

A **Simple Solution** is to go through all windows of every size, find maximum of all windows. Below is C++ implementation of this idea.

```
// A naive method to find maximum of minimum of all windows of
// different sizes
#include <iostream>
```

```
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 = arr[0];

        // 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;
}
```

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 largest element on right side of arr[i] such that the element is smaller than arr[i]. Similarly, previous smaller element is on left side/
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 1, 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
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

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., an[i+1].

b) If ans[i] is not filled it means there is no direct element which is minimum of length i and therefore eith

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 C++ implementation of above algorithm.

```
// 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
    // http://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;
}
```

Output:

```
 70 30 20 10 10 10 10
```

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

This article is contributed by **Ekta Goel**. 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-the-maximum-of-minimums-for-every-window-size-in-a-given-array/

Category: Arrays Tags: stack

Post navigation

← Walmart Labs Interview Experience | Set 5 (On-Campus) Informatica Interview Experience | Set 2 (On-Campus) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.