1. **Implement Queue using Stacks:**

**Method 1 (By making enQueue operation costly)**

This method makes sure that oldest entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

**enQueue(q, x)**

1) While stack1 is not empty, push everything from satck1 to stack2.

2) Push x to stack1 (assuming size of stacks is unlimited).

3) Push everything back to stack1.

**deQueue(q)**

1) If stack1 is empty then error

2) Pop an item from stack1 and return it

**(this operation is more costlier. As first we have to transfer all elements from stack1 to stack2. then add the current element in stack1 and again shift all elements from stack 2 to stack 1. )**

**Method 2 (By making deQueue operation costly)**

In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

**enQueue(q, x)**

1) Push x to stack1 (assuming size of stacks is unlimited).

**deQueue(q)**

1) If both stacks are empty then error.

2) If stack2 is empty

While stack1 is not empty, push everything from stack1 to stack2.

3) Pop the element from stack2 and return it.

Method 2 is definitely better than method 1.

Method 1 moves all the elements twice in enQueue operation, while method 2 (in deQueue operation) moves the elements once and moves elements only if stack2 empty.

**2.Design and Implement Special Stack Data Structure**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.

**Solution:**It could be combination of two stacks (but then pop() would be costly)

It could be combination of Min heap and stack. (then fix time for insertion and deletion O(logn) and getMin can be served in O(1))

However, if pop() does remove the minimum element as well, we need to re think.

**3.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.

**4.Implement Stack using Queues:**

We can do that.

**5.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.

**How to implement a stack which will support following operations in O(1) time complexity?**

1) push() which adds an element to the top of stack.

2) pop() which removes an element from top of stack.

3) findMiddle() which will return middle element of the stack.

4) deleteMiddle() which will delete the middle element.

**Push and pop are standard stack operations.**

**The important question is, whether to use a linked list or array for implementation of stack?**

Please note that, we need to find and delete middle element. Deleting an element from middle is not O(1) for array. Also, we may need to move the middle pointer up when we push an element and move down when we pop(). In singly linked list, moving middle pointer in both directions is not possible.

**The idea is to use Doubly Linked List (DLL).** We can delete middle element in O(1) time by maintaining mid pointer. We can move mid pointer in both directions using previous and next pointers.

Following is implementation of push(), pop() and findMiddle() operations. Implementation of deleteMiddle() is left as an exercise. If there are even elements in stack, findMiddle() returns the first middle element. For example, if stack contains {1, 2, 3, 4}, then findMiddle() would return 2.

**6.Design a stack with following operations:**

a) push(Stack s, x): Adds an item x to stack s

b) pop(Stack s): Removes the top item from stack s

c) merge(Stack s1, Stack s2): Merge contents of s2 into s1.

**Time Complexity of all above operations should be O(1).**

If we use array implementation of stack, then merge is not possible to do in O(1) time as we have to do following steps.

a) Delete old arrays

b) Create a new array for s1 with size equal to size of old array for s1 plus size of s2.

c) Copy old contents of s1 and s2 to new array for s1

The above operations take O(n) time.

**So, we won’t design it with arrays.**

We can use a linked list with two pointers, one pointer to first node (also used as top when elements are added and removed from beginning). The other pointer is needed for last node so that we can quickly link the linked list of s2 at the end of s1. Following are all operations.

a) push(): Adds the new item at the beginning of linked list using first pointer.

b) pop(): Removes an item from beginning using first pointer.

c) merge(): Links the first pointer second stack as next of last pointer of first list.

**Can we do it if we are not allowed to use extra pointer?**

We can do it with **circular linked list**. The idea is to keep track of last node in linked list. The next of last node indicates top of stack.

a) push(): Adds the new item as next of last node.

b) pop(): Removes next of last node.

c) merge(): Links the top (next of last) of second list to the top (next of last) of first list. And makes last of second list as last of whole list.

**7.Design a stack that supports getMin() in O(1) time and O(1) extra space**

**Push(x) : Inserts x at the top of stack.**

If stack is empty, insert x into the stack and make minEle equal to x.

If stack is not empty, compare x with minEle. Two cases arise:

If x is greater than or equal to minEle, simply insert x.

If x is less than minEle, insert (2\*x – minEle) into the stack and make minEle equal to x. For example, let previous minEle was 3. Now we want to insert 2. We update minEle as 2 and insert 2\*2 – 3 = 1 into the stack.

**Pop() : Removes an element from top of stack.**

Remove element from top. Let the **removed element be y**. Two cases arise:

**If y is greater than or equal** to minEle, **y is the minimum element.** the minimum element in the stack is still minEle.

**If y is less than minEle, for the current time minEle is the minimum element .** The minimum element now becomes (2\*minEle – y),This is where we retrieve previous minimum from current minimum and its value in stack. For example, let the element to be removed be 1 and minEle be 2. We remove 1 and update minEle as 2\*2 – 1 = 3.

**Important Points:**

Stack doesn’t hold actual value of an element if it is minimum so far.

Actual minimum element is always stored in minEle.

**Now, suppose, we only need to getMin() at every step. (Building minheap take O(n) time. And every insertion takes O(logn). Every deletion takes O(logn). Now, here, every insertion and deletion comes in O(1))**

Now, this will allow, push, pop and getMin() in O(1)

**How does this approach work?**

When element to be inserted is less than minEle, we insert “2x – minEle”. The important thing to notes is, 2x – minEle will always be less than x (proved below), i.e., new minEle and while popping out this element we will see that something unusual has happened as the popped element is less than the minEle. So we will be updating minEle.

**How 2\*x - minEle is less than x in push()?**

x < minEle which means x - minEle < 0

// Adding x on both sides

x - minEle + x < 0 + x

2\*x - minEle < x

We can conclude 2\*x - minEle < new minEle

(That is why an element can be found as smaller than minElement during pop)

While popping out, if we find the element(y) less than the current minEle, we find the new minEle = 2\*minEle – y.

How previous minimum element, prevMinEle is, 2\*minEle - y

**in pop() is y the popped element?**

// We pushed y as 2x - prevMinEle. Here

// prevMinEle is minEle before y was inserted

y = 2\*x - prevMinEle

// Value of minEle was made equal to x

minEle = x .

new minEle = 2 \* minEle - y

= 2\*x - (2\*x - prevMinEle)

= prevMinEle // This is what we wanted

1. **Implement K Stacks In An Array:**

The idea is to use two extra arrays for efficient implementation of k stacks in an array. This may not make much sense for integer stacks, but stack items can be large for example stacks of employees, students, etc where every item is of hundreds of bytes. For such large stacks, the extra space used is comparatively very less as we use two integer arrays as extra space.

Following are the two extra arrays are used:

1) top[]: This is of size k and stores indexes of top elements in all stacks.

2) next[]: This is of size n and stores indexes of next item for the items in array arr[]. Here arr[] is actual array that stores k stacks.

Together with k stacks, a stack of free slots in arr[] is also maintained. The top of this stack is stored in a variable ‘free’.

All entries in top[] are initialized as -1 to indicate that all stacks are empty. All entries next[i] are initialized as i+1 because all slots are free initially and pointing to next slot. Top of free stack, ‘free’ is initialized as 0.

Following is implementation of the above idea.

// 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;

}

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

}

This is constructor. Now, check the following pointers. First, top is of size k. Because, we are implementing k stacks.

Second thing,next array is of size n and stores indexes of next item for the items in array arr[]. 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.

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

Now, next[n-1]=-1 //because, there are no free slots afterwards

1. **Implement A Stack Using Priority Queue:**

In priority queue, we assign priority to the elements that are being pushed. A stack requires elements to be processed in Last in First Out manner. The idea is to associate a count that determines when it was pushed. This count works as a key for the priority queue.

So the implementation of stack uses a priority queue of pairs, with the first element serving as the key.

1. **Infix To Postfix:**

1. Scan the infix expression from left to right.

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

3. Else,

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

…..3.2 Else, Pop the operator from the stack until the precedence of the scanned operator is less-equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.

4. If the scanned character is an ‘(‘, push it to the stack.

5. If the scanned character is an ‘)’, pop and output from the stack until an ‘(‘ is encountered.

6. Repeat steps 2-6 until infix expression is scanned.

7. Pop and output from the stack until it is not empty.

|  |  |  |  |
| --- | --- | --- | --- |
| **Step No.** | **Symbol** | **Postfix** | **Stack** |
| 1 | ( |  | ( |
| 2 | A | A | ( |
| 3 | + | A | (+ |
| 4 | B | AB | (+ |
| 5 | ) | AB+ |  |
| 6 | \* | AB+ | \* |
| 7 | C | AB+C | \* |
| 8 |  | AB+C\* |  |

**10.Infix To Prefix Operation:**

Step 1. Push “)” onto STACK, and add “(“ to end of the A

Step 2. **Scan A from right to left** and repeat step 3 to 6 for each element of A until the STACK is empty

Step 3. If an operand is encountered add it to B

Step 4. If a right parenthesis is encountered push it onto STACK

Step 5. If an operator is encountered then:

a. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same

or higher precedence than the operator.

b. Add operator to STACK

Step 6. If left parenthesis is encountered then

a. Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encountered)

b. Remove the left parenthesis

Step 7. Exit

**11.The Stock Span Problem:**

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}

// C program for brute force method to calculate stock span values

#include <stdio.h>

// Fills array S[] with span values

void calculateSpan(int price[], int n, int S[])

{

// Span value of first day is always 1

S[0] = 1;

// Calculate span value of remaining days by linearly checking

// previous days

for (int i = 1; i < n; i++)

{

S[i] = 1; // Initialize span value

// Traverse left while the next element on left is smaller

// than price[i]

for (int j = i-1; (j>=0)&&(price[i]>=price[j]); j--)

S[i]++;

}

}

// A utility function to print elements of array

void printArray(int arr[], int n)

{

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

printf("%d ", arr[i]);

}

// Driver program to test above function

int main()

{

int price[] = {10, 4, 5, 90, 120, 80};

int n = sizeof(price)/sizeof(price[0]);

int S[n];

// Fill the span values in array S[]

calculateSpan(price, n, S);

// print the calculated span values

printArray(S, n);

return 0;

}

Now, S[0] will always be 1.

Because, s[i] is defined as the maximum number of consecutive days just before the given day and for 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.

Now, there’s no day before the given day.

Now, it includes the current day. As the price of the stock on the current day is less than or equal to its price on the given day.

We will store the index in stack rather than the price. (obviously, that is the easiest way to track)

**But, this will consume more time.**

**// 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;**

**}**

**12.Check for balanced parentheses in an expression:**

**13.Next Greater Element**I can do it using stack.

**14.Expression Evaluation:**

This is a infix expression evaluation.

1. While there are still tokens to be read in,

1.1 Get the next token.

1.2 If the token is:

1.2.1 A number: push it onto the value stack.

1.2.2 A variable: get its value, and push onto the value stack.

1.2.3 A left parenthesis: push it onto the operator stack.

1.2.4 A right parenthesis:

1 While the thing on top of the operator stack is not a

left parenthesis,

1 Pop the operator from the operator stack.

2 Pop the value stack twice, getting two operands.

3 Apply the operator to the operands, in the correct order.

4 Push the result onto the value stack.

2 Pop the left parenthesis from the operator stack, and discard it.

1.2.5 An operator (call it thisOp):

1 While the operator stack is not empty, and the top thing on the

operator stack has the same or greater precedence as thisOp, **(so, until the thisOp has same or lower precedence than the stack top)**

1 Pop the operator from the operator stack.

2 Pop the value stack twice, getting two operands.

3 Apply the operator to the operands, in the correct order.

4 Push the result onto the value stack.

2 Push thisOp onto the operator stack. **(if there is no operator in the operator stack which has equal or greater precedence as thisOp**)

1. While the operator stack is not empty, **(at the end if operator stack is not empty**

**We have to check if operand stack is not empty as well)**

1 Pop the operator from the operator stack.

2 Pop the value stack twice, getting two operands.

3 Apply the operator to the operands, in the correct order.

4 Push the result onto the value stack.

3. At this point the operator stack should be empty, and the value

stack should have only one value in it, which is the final result.

**15.Postfix Operator Evaluation:**

**16.Reverse A Stack Using Recursion:**

I can do that.

**17.Sort A Stack Using Recursion:**I can do that.

**18.Sorted Stack Using Temporary Stack:**

We follow this algorithm.

Create a temporary stack say tmpStack.

While input stack is NOT empty do this:

Pop an element from input stack call it temp

while temporary stack is NOT empty and top of temporary stack is greater than temp,

pop from temporary stack and push it to the input stack

push temp in temporary stack

The sorted numbers are in tmpStack  
  
**19.Reverse a stack without using extra space in O(n)**

If the stack is internally represented as linked list.

**20.Largest Rectangular Area in a Histogram:**

It’s more about finding next minimum element’s index and previous minimum element’s index fro every element.

**21.Iterative Tower Of Hanoi:**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

**22.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.

**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.

**Method 2 ( O(1) auxiliary space )**

**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.**

1. **If current\_max is not 0, then return -1 to ensure that the parenthesis are not balanced. Else return max**