ï»¿ Tutorial- 1 Stack

What is a stack?

Stack is a simple linear data structure that allows adding and removing elements in a particular order.

The only property that makes a stack stand out is this order.

This Order may be LIFO(Last In First Out) or FILO(First In Last Out).Every time an element is added, it goes on the top of the stack, the only element that can be removed is the element that was at the top of the stack, just like a pile of objects.

Visualisation:

//a diagram of a pile of book here

Now in the above diagram,How are books arranged in this Stack?

These books are analogous to elements/data that we will store in this Stack.

1.Books are kept in one above the another fashion.
2.The first book that was inserted will be taken out at last(clearly seen,because every other book is at the top of it).
3.The last book that was inserted will be taken out at first and will be served(just because it is at the top now).

//diagram of a stack of chairs

Suppose at your home you have multiple chairs then you put them together to form a vertical pile. From that vertical pile the chair which is placed last is always removed first.

Chair which was placed first is removed last.

In this way we can see how stack is related to us.

Now upto these visualisations might have got straight into the head aboout the stack,this is what a stack is.

Now we will move on to the operations that can be performed on a stack.

Operations:

The following basic operations
Push:Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
Pop:Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
Peek or Top: Returns top element of stack.
isEmpty:Returns true if stack is empty, else false.
The above operations will have to be performed in order to solve a problem and this will be illustrated in the sample problems that we are about to witness in the following sections.

But before moving on to discussion of problems,we would obviously want to implement a stack(We will use C++ here).

Algorithm for PUSH Operation
A simple algorithm for Push operation can be derived as follows :
The process of putting a new data element onto stack is known as a Push

Operation. Push operation involves a series of steps âˆ’
Step 1:âˆ’ Checks if the stack is full.
Step 2:âˆ’ If the stack is full, produces an error and exit.
Step 3:âˆ’ If the stack is not full, increments top to point next empty space.
Step 4:âˆ’ Adds data element to the stack location, where top is pointing.
Step 5:âˆ’ Returns success.
PSEUDO CODE:

```
begin procedure push: stack, data
    if stack is full
         return null
    endif
    top â†� top + 1
    stack[top] â†� data
end procedure
```

Actual Implementation of Push operation:

```
//diagram here to be put later on.

void push(int data)
{
if(!isFull()) //we ll define isFull in the later sections.
{
top = top + 1;
stack[top] = data;
}
else
{
printf("Could not insert data, Stack is full.\n");
}
}
```

Now lets move on to pop operation.
Algorithm for POP Operation:
Accessing the content while removing it from the stack, is known as a Pop Operation.
In the implementation of pop operation(in particular,the array implemetation), the data element is not actually removed, instead top is decremented to a lower position in the stack
to point to the next value.
A Pop operation may involve the following steps âˆ’
Step 1âˆ’ Checks if the stack is empty.
Step 2âˆ’ If the stack is empty, produces an error and exit.
Step 3âˆ’ If the stack is not empty, accesses the data element at whichÂ topÂ is pointing.
Step 4âˆ’ Decreases the value of top by 1.
Step 5âˆ’ Returns success.

PSEUDO CODE for pop operation.

```
begin procedure pop: stack
     if stack is empty
        return null
    endif
    data â†� stack[top]
    top â†� top - 1
    return data
end procedure
```

Actual Implementation in C++

```
int pop(int data)
{
    if(!isempty())
   {
       data = stack[top];
       top = top â€“ 1;
       return data;
    }
else
   {
       printf("Could not retrieve data, Stack is empty.\n");
    }
 }
```

//put a picture for pop oper.

Implemnt for isEmpty oper.

```
bool Stack::isEmpty()
{
return (top < 0);
}
bool isempty() {
if(top == -1)
  return true;
else
  return false;
 }
```

begin procedure isempty

Implementation of isfull() function

```
bool isfull() {
if(top == MAXSIZE)
 return true;
else
 return false;
}
```

//mend these later.

Analysis of Stacks
Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.
Push Operation: O(1)
Pop Operation: O(1)
Top Operation: O(1)
Search Operation: O(n)

STL Stack:

A stack can be created using STL in C++ as described below.

```
stack<int> s;
// pushing elements into stack
s.push(2);
s.push(3);
s.push(4);
cout << s.top(); // prints 4, as 4 is the topmost element
cout << s.size(); // prints 3, as there are 3 elements in stack.
```

Accordingly,various other functions can be given as s.pop(),s.top() etc,which will be described below.

We will start examples now to get an overview of how to solve problems:

first problem:

1.histogram problem.
2.Parenthesis problem.
3.Signal tower.
4.Infix to postfix?doubt over this.

Histogram problem:

problem statement//

Largest rectangle in a histogram:Given n non-negative integers representing the
histogramâ€™s bar heights where the width of each bar is 1, find the area of
largest rectangle in the histogram.

//diagram for the histogram here to be put.

If you want to try out this problem before looking at the solution, you can do
that in various platforms like SPOJ.

//provide the link for the SPOJ problem here.

We will try to explain the process of HOW TO COME UP WITH A SOLUTION.

The first problem that we have to face is:
which rectangles should we choose?is there any special property that these
special rectangles have?
1. The ones which cover a contiguous range of the given input histogram and
their width has to be interger of course,because there is no point in covering a
fraction of a bar,and whose height equals the minimum bar height in the range
height[i..j].
2. This is because the rectangle height can't exceed the minimum height in the
range,secondly there is no point in choosing a height below the minimum,coz we
can always increase it to the minimum height to get a bigger area.right?

This limits the set of rectangles we need to consider. Formally, we need to
consider only those rectangles with width = j-i+1 (0â‰¤iâ‰¤j<n) and height =
min(height[i..j]).

We can directly implement this solution.There are only n^2 choices for i and j.
If we calculate the minimum height in the range [i..j], this will have time
complexity O(n^3). Instead, we can even keep track of the minimum height in the
inner loop for j,that can be implemented with O(n^2) time complexity.

```
int largestRectangleArea(vector<int> &A)
{
     int maxArea = 0;
     for (int i = 0; i < A.size(); i++) {
           for (int j = i, mn = A[i]; j < A.size(); j++)
                {
                     mn = min(mn, A[j]);
                     maxArea = max(maxArea, (j-i+1) * mn);
               }
}
 return maxArea;
 }
```

We are still doing a lot of repeated work by considering all n^2 rectangles.
There are only n possible heights. For each position j, we need to consider only
1 rectangle: the one with height = height[j] and width = k-i+1, where 0â‰¤iâ‰¤jâ
‰¤k<n, height[i..k] â‰¥ height[j], height[i-1] < height[j] and height[k+1] <
height[j].
In the above example, for index 3, we need only consider the rectangle with
height = 4 and width = 3 ([5, 4, 5]).
Put simply, we need to grow the rectangle from j towards left and right while
the bar heights are greater than or equal to height[j]. If we do this naively,

weâ€™ll still end up with O(n^2) time complexity. If for any index j, we could find, in better than linear complexity, the first indices i (while going left from j) and k (while going right from j) where the bar height becomes smaller than height[j], we could solve this problem in complexity better than O(n^2).

Thereâ€™s a trick to do this with a stack. We can precompute the left end of the rectangle situated at position i and having height = height[i] as follows:

```
vector posLeft(n);
stack S;
for (int i = 0; i < n; i++) {
  while (!S.empty() && height[S.top()] â‰¥ height[i])
    S.pop();
  posLeft[i] = S.empty() ? 0 : S.top() + 1;
  S.push(i);
}
```

Each index is pushed onto the stack and popped off the stack only once, so the overall time complexity of this procedure is O(n). To get an intuition for what it is doing, observe that each bar gets rid of the higher bars on the left of it so that no bar on the right of it need to consider those indices.
Similarly, we can precompute the right end of the rectangle for each position i. This gives us the following implementation with O(n) space and time complexity:

```
int largestRectangleArea(vector<int> &A) {
int n = A.size();
vector<int> posLeft(n);
stack<int> S;
for (int i = 0; i < n; i++) {
while (!S.empty() && A[S.top()] >= A[i])
S.pop();
posLeft[i] = S.empty() ? 0 : S.top() + 1;
S.push(i);
}
Â
vector<int> posRight(n);
while (!S.empty()) S.pop();
for (int i = n-1; i >= 0; i--) {
while (!S.empty() && A[S.top()] >= A[i])
S.pop();
posRight[i] = S.empty() ? n-1 : S.top() - 1;
S.push(i);
}
Â
int maxArea = 0;
for (int i = 0; i < n; i++)
maxArea = max(maxArea, (posRight[i] - posLeft[i] + 1) * A[i]);
return maxArea;
}
```

This should be a perfectly acceptable solution but we can do further optimizations. Observe that we donâ€™t need to precompute and store both left and right ends. If we precompute the right ends, we can iterate from the left and
compute the areas while computing the left ends (or we can do it the other way around).

```
int largestRectangleArea(vector<int> &A) {
int n = A.size();
vector<int> posRight(n);
stack<int> S;
for (int i = n-1; i >= 0; i--) {
while (!S.empty() && A[S.top()] >= A[i])
S.pop();
posRight[i] = S.empty() ? n-1 : S.top() - 1;
S.push(i);
}
Â
while (!S.empty()) S.pop();
int maxArea = 0;
for (int i = 0; i < n; i++) {
while (!S.empty() && A[S.top()] >= A[i])
S.pop();
long long area = A[i] * (posRight[i] + 1 - (S.empty() ? 0 : S.top() + 1));
if (area > maxArea)
maxArea = area;
S.push(i);
}
return maxArea;
}
```

Oh but thereâ€™s still more optimizations that we can do. Observe when an index gets popped off the stack. This happens because we find the first index (on the right of it) which has a smaller height. This already gives us the right end of the rectangle. So we donâ€™t need to precompute either the left or the right ends of the rectangle. Unfortunately, the space and time complexity is still O(n) but the solution is much prettier!

```
int largestRectangleArea(vector<int> &A) {
A.push_back(0); // sentinel to ensure all indices get popped off the stack
stack<int> S;
int maxArea = 0;
for (int i = 0; i < A.size(); i++) {
while (!S.empty() && A[S.top()] >= A[i]) {
int height = A[S.top()];
S.pop();
int left = S.empty() ? 0 : S.top() + 1, right = i - 1;
maxArea = max(maxArea, (right - left + 1) * height);
}
S.push(i);
}
return maxArea;
}
```

//start discussion of first problem:string parenthesis.

//similarly two other examples...

Other uses:

While it may seem that stacks are rarely implemented explicitly, a solid understanding of how they work, and how they are used implicitly, is worthwhile education. Those who have been programming for a while are familiar with the way the stack is used every time a subroutine is called from within a program. Any parameters, and usually any local variables, are allocated out of space on the stack. Then, after the subroutine has finished,
the local variables are removed, and the return address is "popped" from the stack, so that program execution can continue where it left off before calling the subroutine.

An understanding of what this implies becomes more important as functions call other functions, which in turn call other functions. Each function call increases the "nesting level" (the depth of function calls, if you will) of the execution, and uses increasingly more space on the stack. Of paramount importance is the case of a recursive function. When a recursive function continually calls itself, stack space is quickly used as the depth of recursion increases.

Video links to be added later.like my code school,tushar roy.
//put the selected problem links from spoj/codechef/codeforces.(only select 10 from these 3).