Stacks

Manoj Gupta

August 15, 2016

One of the basic thing that a C compiler checks while compiling your program is whether the parenthesis in your program is balanced. If they are not then the compiler should flag a compilation error. Let us abstract out this problem faced by the compiler. Let S be a string that has element only from the set { '{', '} '(', ')' }. Elements in the set { '{', '(' } are called the opening element. The first element of this set is called opening parenthesis while the last is called an opening bracket. Similarly, we can define closing element, closing parenthesis and closing brackets symmetrically.

Definition 0.1. *A string S is said to be balanced if*

1. *For each closing element there is exactly one opening element.*
2. *The elements are properly nested.*

For example the the following strings are balanced.

$$\{\}()\{()\{\}\}, (\{\})(), \{(()\{\}))\}$$

And the following strings are not balanced.

$$()\{, \{()(\}, ()((())\{\}$$

Consider the following problem.

Question 1. *Given a string, decide whether it is balanced.*

Let us assume that the string is represented using an array. A simple algorithm for the above problem is to find the first closing element, say at array index i. Then observe that the opening element of the same type should lie at index $i-1$. If not, then we can conclude that the string is not balanced. Otherwise, we can delete both the elements and continue with our procedure given in Figure 4.

This procedure just tries to mimic the above method, i.e., find the first closing element, say at index i and then compare it with the opening element that is to its left. However, the opening element now may not lie at index $i-1$. This is because we might have removed elements in the previous iterations. So, after finding index i, we must find a non-empty index j such that $j < i$.

One can show that the algorithm below indeed finds if a parenthesis is balanced. We focus on the running time of our algorithm. The worst case instance for our algorithm is as follows: $S[i] =$ '(' for $1 \leq i \leq n/4$ and its matching closing brackets lie at $3n/4 \leq i \leq n$. From $n/4+1$ to $3n/4-1$, there are $n/4$ strings of type '{}'. Consider the run of the algorithm on this input. Assume that we have processed all the closing (and opening) parenthesis. So our algorithm then finds the first bracket at index $3n/4$. The matching closing bracket is at location $n/4$. So our algorithm has to check all location from $3n/4$ to $n/4$. So, the time taken to find the opening bracket is $n/2 = O(n)$. In fact for all $i > 3n/4$, to find the corresponding opening bracket is $O(n)$. Since there a $n/4$ closing bracket, the total time taken by our algorithm is at least $O(n^2)$.

```
  /* Input:  S is an array which contains the string                    */
1 while S contains a closing element do
2 |   Let i be the location of first closing element;
3 |   Let j < i be the first non-empty location to the left of i;
4 |   if S[j] is a opening element of the same type as the closing element S[i] then
5 |   |   remove the opening and closing element from S[j] and S[i] respectively ;
6 |   end
7 |   else
8 |   |   print("Not a balanced parenthesis");
9 |   end
10 end
11 if Array S contains a non-empty string then
12 |   print("Not a balanced parenthesis");
13 end
```

Figure 1: FIND-BALANCED(S)

Can we find an algorithm with running time $O(n)$? In the above algorithm, a lot of time was wasted in accessing empty locations in the array. We want to avoid it as this was the main reason for our under performance. Can we avoid it? In particular, can we do the following?

Question 2. *Given a closing element at location i, can we find the location $j < i$ in $O(1)$ time.*

Note that in our algorithm, the opening element is not processed till its corresponding closing element is found. This can be viewed as follows: we just put all the opening element is some data-structure one by one. Once we find a closing element, we remove one opening element from this data-structure. The crux of the algorithm lies in finding which element should be removed. Our algorithm in Figure 4 suggests that opening elements should be removed in the following order:

Observation 0.2. *The last element inserted in the data-structure is the first element to be removed.*

Such a requirement is called LIFO - Last in First Out. We now describe a data-structure – called stack – that supports the following operations:

```
1 top ← 0;
```

(a) CREATE-EMPTY()

```
1 if top < n then
2 |   top ← top + 1;
3 |   A[top] ← a;
4 end
5 else
6 |   print "Stack Full";
7 end
```

(b) PUSH(a)

```
1 if top = 0 then
2 |   print "Stack Empty";
3 end
4 else
5 |   a ← A[top];
6 |   top ← top − 1;
7 |   return a;
8 end
```

(c) POP()

Figure 2: Stacks Using an Array

1. CREATE-EMPTY()

   Creates an empty stack.

2. PUSH(a)

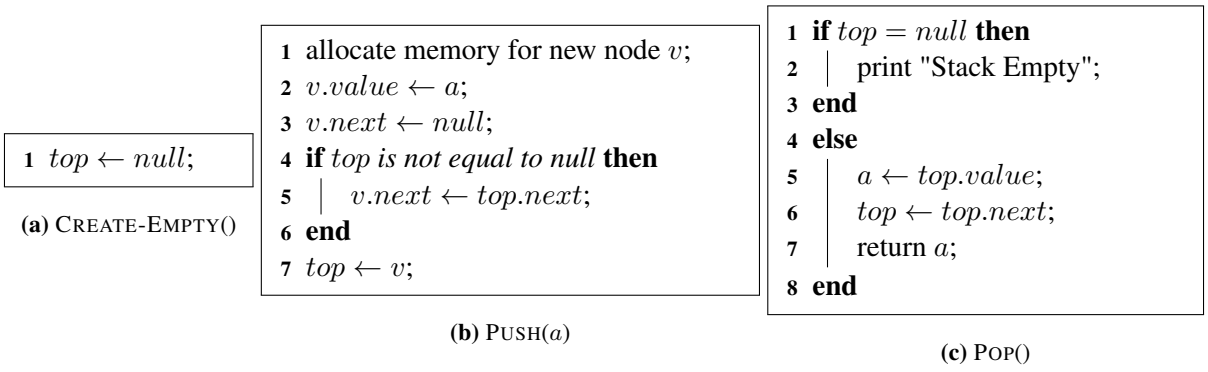
   Push an element a on the top of the stack.

3. POP()

   Pop an element from the top of the stack. This element is the last element to be pushed on to the stack.

Stacks can be implemented using arrays and linked list. Let us first implement it using an array $A[1 \ldots n]$. We will also have another variable top which is the index of the top of the stack.

The complete pseudo-code for implementing stack using an array is given in Figure 2. When an element a is pushed on to the stack, top is incremented by 1. And for POP(), top is decremented by 1. One can check that the time taken by all the three procedures: CREATE-EMPTY(), PUSH(a) and POP() is $O(1)$. The only problem with the above procedure is that the size of the stack is bounded by n which is fixed at the definition of the array. We now describe an alternate procedure which uses linked list in Figure 3.



(a) CREATE-EMPTY()

(b) PUSH(a)

(c) POP()

Figure 3: Stacks Using a Linked List

We are now ready to design an elegant algorithm for balanced parenthesis using Stacks. The algorithm will try to mimic our trivial algorithm in Figure 4. Our algorithm puts all the opening element in the stack. Whenever a closing element is encountered, we just pop the top element from the stack. This steps mimics the step in the trivial algorithm that finds the nearest opening element to the left of i. Thus we don't have to move through the array which was wasteful. In our new algorithm this can be done in $O(1)$ and not $O(n)$ as done in the trivial algorithm. Thus the running time of our algorithm is $O(n)$.

```
  /* Input:  S is an array which contains the string                    */
1 for i ← 1 to n do
2 |   if S[i] is an opening element then
3 |   |   PUSH(S[i]);
4 |   end
5 end
6 else
7 |   a ← POP();
8 |   if a = '(' and S[i] =')' or a = '{' and S[i] = '}' then
9 |   |   continue;
10|   end
11|   else
12|   |   print "Not a balanced parenthesis" ;
13|   end
14 end
15 if Stack is not empty then
16 |   print "Not a balanced parenthesis";
17 end
```

Figure 4: FIND-BALANCED(S)