Data Structures
Spring 2009

Kishore Kothapalli

Chapter 4

The Stack and the Queue

4.1 A Gentle Introduction

The array as a data structure did not have a rich repertoire of operations beyond accessing the ith indexed element. However, we will see that there are several applications where a preimposed access pattern on the arrays are required. In this chapter, we will see two such data structures: stacks and queues. We will study the stack and the queue as an abstract data type or a data structure and also see example applications that use these data structures.

4.2 The Stack Data Structure

The stack data structure is characterized by its last-in-first-out access mechanism. Thus, the element that is added most recently will be the element that is removed first on a remove call. Such an order of addition and deletion of items occurs in several natural settings. Consider for example, a stack of plates in a cafeteria. The first plate that is taken out is the plate that is on the top of the pile. This plate would be the one that is added to the pile most recently.

Another example could be as follows. We are all used to text editors that allow for undo and redo operations. In such an editor, suppose S is the present text and a word $w1$ is deleted to get S'. Then a word $w2$ is deleted. Now, the user wishes to undo some of the changes. To get the correct result, it is possible only if all the operations done after $w1$ is deleted are also undone. Moreover, these operations must be performed in their reverse order. Thus, we need a mechanism to associate a stored set of items.

Irrespective of the implemenation mechanism, the stack data structure has the following basic operations. Let S be a stack.

- `create` : Create an empty stack.
- `push`(element) : Pushes the item `element` to the top of the stack.
- `pop()` : Returns the item that is most recently pushed.
- `size()` : Return the size of the stack.

Let us consider how to implement a stack using an array. Let S be an array of size n. To support these two operations, it is useful to have an index called `top` that points to the element that is most recently added. The following is a possible implementation of the operation `push`(element).

```
Algorithm Push(element)
begin
    top = top + 1;
    $S$[top] = element;
end;
```

Notice however that the above algorithm runs into a trouble when the value of `top` exceedes the size of the array. For this purpose, we modify the above algorithm to do some error handling. Typically, when one is trying to push another element to an already full stack, an error message can be returned. The following pseudocode does that.

```
Algorithm Push(element)
begin
    if (top == $n$) then
        return ”ERROR: STACK FULL”;
    top = top + 1;
    $S$[top] = element;
end;
```

Similarly, the operation `pop` can be implemened as follows. In this case, it would be an error if one tries to pop from an empty stack.

```
Algorithm Pop()
begin
    if (top == $0$) then
        return ”ERROR: STACK EMPTY”;
    return $S$[top–];
end;
```

An example of operations is shown below. Let S be an empty stack of integers. Upon calling Push(6), Push(4), Push(9), the stack would have three items and top equals 3. A pop() returns 9, another push(7) adds 7 to to top of 4. A pop then returns 7.

4.3 Applications of Stacks

4.3.1 Expression Evaluation

One of the prominent applications of stacks is to expression evaluation. Consider a table calculator which evaluates arithemetic expressions involving addition, multiplication, subtraction, and division. For example, 2+3 * 5 - 7 is a valid expression. The result of the above expression is 10 as multiplication has precedence over addition and subtraction. To disambiguate, one also uses parantesis and write the same expression as 2 + (3*5) - 7. However, it would be quite cumbersome to use parantheses when especially, the precedence is known.

Hence, one needs to first convert a given expression into an non-ambiguous model so that evaluation can be done easily. There are three ways to write an expression. The above way of writing expressions is called the infix notation because the operators are placed in between the two operators. There are other ways of writing an expression. In the prefix notation, operators preceede the operands. So the above expression would be written as -+*3 5 2 7. In the postfix notation, the operators are written after the operands. The postfix equivalent way of writing the above expression would be 3 5* 2 + 7 -. It turns out that the postfix and prefix notations are free of ambiguity. We will how that is the case first and then see how to convert a given expression in the infix form to its prefix form.

⟦***Lab: infix to postfix and evaluation***⟧

Evaluating a Postfix Expression

Consider an expression given in the postifix notation. We now see how to evaluate such an expression. For example, ab * c + is a postfix expression. Since the operators follow the operands, it is intuitive to see if the previously available two operands are the corresponding operands for a given operator. This intuition serves well and is correct.

So, when processing a postfix expression from left to right, when we encounter an operator, we have to apply the operation to the two most recent operands. This suggests that the operands should be placed on a stack. The following pseudocode uses the above idea. Let E be a postfix expression with size n.

```
Algorithm EvaluatePostfix($(E)$)
begin
    Stack S;
    for $i = 1$ to $n$ do
    begin
        if $E[i]$ is an operator, say $o$ then
            operand1 = S.pop();
            operand2 = S.pop();
            value = operand1 $o$ operand2;
            S.push(value);
        else
            S.push($E[i]$);
    end-for
    End
```

Consider the expression $a\ b\ +\ c\ d\ *\ e\ f\ +\ +\ +$. The evaluation of the above expression is shown below.

Input Symbol	Stack	Comment
a	a	Push a to the stack
b	$a\ b$	Push b to the stack
+	v_1	a, b popped, $v_1 = a + b$ pushed
c	$v_1\ c$	
d	$v_1\ c\ d$	
*	$v_1\ v_2$	c, d popped, $v_2 = c * d$ pushed
e	$v_1\ v_2\ e$	
f	$v_1\ v_2\ e\ f$	
+	$v_1\ v_2\ v_3$	e, f popped, $v_3 = e + f$ pushed
+	$v_1\ v_4$	v_2, v_3 popped, $v_2 = v_3 + v_4$ pushed
+	EMPTY	$v_1 + v_4$ returned

Convering an Expression from Infix to Postfix

This brings us to the next question : How to convert a given infix expression to its equivalent postfix form. The infix expression contains parantheses apart from operators and operands. While converting to a postfix expression, we have to worry about the precedence of the operators. To understand this issue, consider the simple expression $a\ +\ b\ *\ c\ + d$. (The correct evaluation is $a + (b * c) + d$.) Now, when the operator $*$ is encountered in a left-to-right scan, only one of the operands (b) is available. The other operand c is yet to be encountered. So, it suggests that firstly, we should store the operands seen so far so that the operands are properly selected. This suggests that one may use a stack for storing the operands. However, one has to be careful to get the correct sequence of operators and the operands. Now we see how to do that.

When scanning an expression from left to right, it is customary to associate the recent operands according to operator precedence. So, let us start by adding operators to the stack and printing out the operands immediately. So we have a in the output and $+$ in the stack. Let us treat '(' as a special operator with the highest precedence. So, we add '(' also to the stack. Now the stack has $+\ ($ with '(' at the top. The operand 'b' is written out. On seeing the operator $*$, we add it to the stack. Now the operand 'c' is printed out. When scanning the ')', we notice that this indicates a completion of an operation. So we should produce operators till the matching '(', which need not be printed. So the output contains $a\ b\ c\ *$. On reading the '+' operator, it is placed in the stack. The operand 'd' is printed to the output. Finally, at the end of the input, we can empty the stack to get the expression $a\ b\ c\ *\ d\ +\ +$.

Notice that when the precedence of operators is well understood, we may not need the parantheses at all. The general algorithm is as follows. If we encounter an operator that has a higher precedence than the operator presently at the top of the stack, then we add the current operator to the stack. Otherwise, we pop from the stack operators that have a higher precedence. If the infix expression has any parantheses, the left parantheses is added to the stack when it is enountered, and when it is popped, it is not printed to the output.

The pseudo-code for this scheme is given below.

TODO

4.3.2 Support for Recursion

For the tutorial and lab implementation, or homework.

4.4 The Queue Data Structure

In the previous section, we have seen a new way of access model to the array can result in good solutions to important applications. A queue is another such data strucure which supports operations such as `insert` and `delete`. One can think of analogies to several situations such as a queue at a ticket reservation office, an operating system job queue, or a queue of aeroplanes ready to take off. In all such settings, the element or the object that is first inserted is the first one to come out of the queue also. Thus, the queue is a First-In-First-Out (FIFO) data structure. This suggests that there are two quantities associated with a queue, the front and the rear. The rear indicates the position where the new elements would be added. The front indicates the position from where elements can be deleted from the queue. Figure **??** illustrates this order.

Formally, a queue is a data structure which supports the following operations.

- `insert`: Insert an element to the rear of the queue
- `delete`: Delete an element from the front of the queue.
- `size`: Return the size of the queue

One can use an array to implement a queue as follows. The quantities front and rear can be kept as auxillary vairables, initialized to 0 in each case. The pseudo-code for the operations is given below. The additional variable size indicates the number of elements in the queue and can be used for error handling purposes also.

```
Algorithm Insert($x$)
begin
    if rear == MAXSIZE then return ERROR;
    Queue[rear] = x;
    size = size + 1;
    rear = rear + 1;
end
```

```
Algorithm Delete()
begin
    if size == 0 then return ERROR;
    size = size - 1;
    return Queue[front++];
end
```

For example, starting from an empty queue, the operations Insert(5), Insert(4), Insert(3), Delete(), Delete(), would result in the following queue states: 5, and 5,4 and 5,4,3, and 4,3, and finally 3.

Notice however that once elements are deleted and front points to an index i, the first $i - 1$ cells are left unused. Once, an element is deleted, the cell where that element is stored is left unused. To overcome this problem, one also uses a *circular queue*. In this implementation, a queue is redered to be full only when all the array cells are currently occupied. More details in the tutorial and the lab.

4.5 Applications of the Queue Data Structure

Let us now study a few applications of the queue data structure. Queues, as mentioned earlier, are used in settings where there is a first-in-first-out style of data processing. Let us develop such applications in this section.

4.5.1 The Packet Queue

TODO

4.5.2