Data Structure and Algorithm handout 2

Apurba Sarkar

August 4, 2017

1 Motivation

As a computer scientist we always try to come up with efficient algorithm for a given problem and one of the basic technique for improving the algorithm is to structure the data in such a way that required operations can be carried out efficiently. In this handout we will discuss about the most basic and commonly data structures.

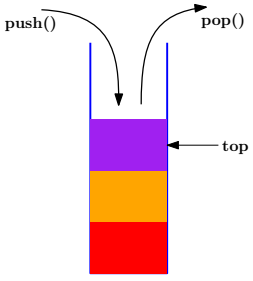
2 Stacks and Queues

One of the most common forms of data organization in computer programs is the ordered or linear list, which is often written as A $= (a_i, a_2, \ldots, a_n)$. The a's are referred to as atoms and they are chosen from some set. The null or empty list has $n = 0$ elements.

Definition 1. *A stack is an ordered list in which all insertions and deletions are made at one end, called the top.*

Definition 2. *A queue is an ordered list in which all insertions take place at one end, the rear, while all deletions take place at the other end, the front.*

The simplest way to represent a stack is by using a one-dimensional array, say STACK$(0 : n - 1)$, where n in the maximum number of elements. The first or bottom element in the stack will be stored at STACK(1), the second at STACK(2) and the i^{th} at STACK(i). Associated with the array will be a variable, typically called *top*, which points to the *top* element in the stack. To



(a) A Stack

(b) A Queue

Algorithm 1: PUSH(ITEM)

```
   Input: ITEM: the item to be
          pushed;
   // Push an element on to the
      Stack, returns true if
      successful
   // else returns false
1  if top ≥ n then
2  |  printf("stack full");
3  |  return false;
4  else
5  |  top = top + 1;
6  |  stack(top) = ITEM;
7  |  return true;
```

Algorithm 2: POP()

```
   // Pops the top element from
      the Stack,
   // returns the top element
      if stack is
   // not empty else returns
      NULL
1  if top < 0 then
2  |  printf("stack empty");
3  |  return NULL;
4  else
5  |  ITEM = stack(top) ;
6  |  top = top − 1;
7  |  return ITEM;
```

test if the stack is empty we ask "if $top = 0$". If not, the top-most element is at STACK(top). Two of the most important operations are inserting(push) and deleting(pop). Algorithm 1 and Alg. 2 explains the push() and pop() operations respectively.

Perhaps the most common occurrence of queues in Computer Science is for the scheduling of jobs. In batch processing, the jobs are queued up as they are read and executed one after another in the order they are received. There are two distinct ends in a queue, the end in which addition takes place is called the ***rear*** and the end in which deletion takes place is called the ***front***. So, when a new job is submitted for execution it joins the queue at the ***rear*** end and the job which is at the ***front*** is the one to be executed next.

The most natural way to represent a queue is a one dimensional array Q($1 : n$) and two variables ***front*** and ***rear***. The convention we shall adopt for these two variables are that ***front*** always 1 less than the actual front of the queue and ***rear*** always points to the last element. Thus, when ***front*** = ***rear***, it signifies that the queue is empty. ***front*** and ***rear*** are initialized to zero, i.e., ***front*** = ***rear*** = 0. Algorithm. 3 and Alg. 4 explains respectively the procedure for adding an element to the queue and deleting an element from the queue.

One important observation in this implementation is that unless the ***front*** regularly catches up with the ***rear*** and both pointers are reset to zero, then the "queue full" signal does not necessarily mean that there are n elements in the queue. In this implementation the queue will gradually move to the right. To resolve this every time "queue full" is signaled, the entire queue needs to be shifted towards left such that the first element is again at Q(1) and ***front***=0. This is very time consuming process specially, when there are many element in the queue at the time when "queue full" is signaled.

Let us take an example to see what could happen in the worst case, if each time the queue becomes full, we choose to move the entire queue to the left so

Algorithm 3: ADDQ(ITEM)

```
   // inserts an element into
      the queue
1  if rear = n then
2  |   printf("Queue full");
3  |   return;
4  rear = rear + 1;
5  Q(rear) = ITEM;
6  return;
```

Algorithm 4: DELETQ()

```
   // deletes an element from
      the queue
1  if front = rear then
2  |   printf("queue empty");
3  |   return;
4  front = front + 1;
5  ITEM = Q(front);
6  return ITEM;
```

that it starts at Q(1). Assume that there are n jobs $J_1, J_2, J_3, \ldots, J_n$ in the queue and we next receive alternate requests to delete and add elements. Each time a new element is added, the entire queue of $n-1$ element is moved left.

front	***rear***	Q(1)	(2)	(3)	.....	(n)	next operation
0	n	J_1	J_2	J_3	.....	J_n	initial state
1	n		J_2	J_3	.....	J_n	delete J_1
0	n	J_2	J_3	J_4	.....	J_{n+1}	add J_{n+1}
1	n		J_3	J_4	.....	J_{n+1}	delete J_2
0	n	J_3	J_4	J_5	.....	J_{n+2}	add J_{n+2}

Figure 2: Example Queue Operations

A more efficient queue representation is obtained by regarding the array Q(1 : n) as circular. It now becomes more convenient to declare the array as Q(0 : $n-1$). When $rear = n-1$ the next element is inserted at Q(0) if that space is free. We use the same convention for the variable ***front*** as before, i.e., ***front*** will always point one position counterclockwise from the first element in the queue. Again, the empty queue is signaled by ***front*** = ***rear***. The variables ***front*** and ***rear*** are both initialized to 1 (i.e., ***front*** = ***rear*** = 1). Figure 3 illustrates some of the possible configurations for a circular queue containing four jobs $J_1 - J_4$. As we have assumed the array Q to be circular, the algorithm for adding and deleting needs to be changed slightly. While adding an elements, the rear should be moved one position clockwise, i.e.,

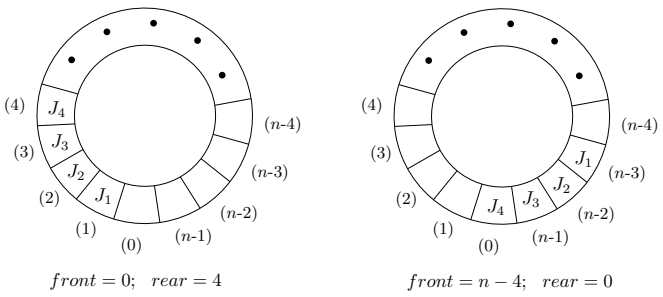


Figure 3: Circular queue.

Algorithm 5: ADDQ(ITEM)

```
   // inserts an element into
      the queue
1  rear = (rear + 1)mod n if
     front = rear then
2  |  printf("Queue full");
3  |  return;
4  Q(rear) = ITEM;
5  return;
```

Algorithm 6: DELETQ()

```
   // deletes an element from
      the queue
1  if front = rear then
2  |  printf("queue empty");
3  |  return;
4  front = (front + 1)mod n;
5  ITEM = Q(front);
6  return ITEM;
```

$$\textbf{if } rear = n - 1 \textbf{ then } rear = 0 \\ \textbf{else } rear = rear + 1$$

The same operation can be easily obtained by using the modulo operator as $rear = (rear + 1)\textbf{mod } n$ Similarly, while deleting an element from the queue, ***front*** should be moved one position clockwise. This can again be achieved easily by modular operation, i.e., $front = (front + 1)\textbf{mod } n$. It is to be observed that assuming the array to be circular, addition and deletion operation can be performed in constant (O(1)) amount of time.

One important and surprising point about the two algorithm is that the test for queue full in ADDQ and the tets for queue empti in DElETQ are the same. However, in the case of ADDQ, when $front = rear$ there is actually one space free, i.e., Q($rear$), since the first element in the queue is not at Q($front$) but is one position clockwise from this point. If we insert an element there (i.e., at Q($rear$)) then we will not be able to distinguish between the cases full and empty, since that will leave $front = rear$. To avoid this we permit a maximum of $n - 1$ elements rather than n elements. We can make use of all n position with the help of a $flag$ variable to distinguish between the two situations, i.e., $flag = 0$ if and only if the queue is empty and $flag = 1$ otherwise. But introduction of this $flag$ variable would slow down the ADDQ and DElETQ algorithms as they will be used many times in any problems involving queue. We can easily trade this one space away for better performance.

3 Application of Stacks and Queues

In this section we will see application of stacks and queues. There are lot of applications these data structure in computer science, we will be discussing few of them.

3.1 Evaluation of Arithmetic Expression

An arithmetic expression is made up of operands, operators and delimiters. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are

- **Infix Notation:** The operators are placed in-between operands in this notation. For example, $a + b * c$. It is easy for us humans to read, write, and speak in infix notation but the same is not so easy with computers.

- **Prefix (Polish) Notation:** In this notation, operator is prefixed to operands, i.e. operator is written ahead of operands. For example, $+a*bc$. This is equivalent to its infix notation $a + b * c$. Prefix notation is also known as Polish Notation.

- **Postfix (Reverse-Polish) Notation:** In this notation, the operator is postfixed to the operands i.e., the operator is written after the operands. For example, $abc * +$. This is equivalent to its infix notation $a + b + c$. This notation is also known as Reverse Polish Notation.

Consider a situation where you are writing a programmable calculator. If the user types in $10 + 10$, how would you compute a result of the expression? You might have a solution for this simple expression. But what if he types in $a/b^c + d * e - a * c$? What would you do then?

There's no need to think about how you're going to compute the result because no matter what you do, it won't be the best algorithm for calculating expressions. That is because there is a lot of overhead in computing the result for expressions which are expressed in this form; which results in a loss of efficiency. The expression that you see above is in Infix Notation. It is a convention which humans are familiar with and is easier to read. But for a computer, calculating the result from an expression in this form is difficult. Hence the need arises to find another suitable system for representing arithmetic expressions which can be easily processed by computing systems. The Prefix and Postfix Notations make the evaluation procedure really simple.

But since we can't expect the user to type an expression in either prefix or postfix, we must convert the infix expression (specified by the user) into prefix or postfix before processing it.

This means that your program would have to have two separate functions, one to convert the infix expression to post or prefix and the second to compute the result from the converted expression.

The first problem with understanding the meaning of an expression is to decide in what order the operations are carried out. To decide the order we need to take care of operator precedence and associativity.

- **Precedence:** When an operand is in between two different operators, which operator will take the operand first, is decided by the *precedence* of an operator over others. For example, $a * b + c \longrightarrow a + (b + c)$. As multiplication operation has precedence over addition, $b * c$ will be evaluated first.

- **Associativity:** Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression $a + b - c$, both $+$ and $-$ have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both $+$ and $-$ are left associative, so the expression will be evaluated as $(a + b) - c$.

Precedence and associativity determines the order of evaluation of an expression. This means that every language must uniquely define such an order. Following is an operator precedence and associativity table Now that we have specified

Sl No	***Operators***	***Precedence***	***Associativity***
1	$\wedge$, unary $-$, unary+	6	right to left
2	*,/	5	left to right
3	+, -	4	left to right
4	$<, \leq, \geq, =, \neq, \not>, \not<$	3	left to right
5	and	2	left to right
6	or	1	left to right

Figure 4: Precedence and Associativity

priorities and rules for breaking ties we know how $a/b \wedge c + d * e - a * c$ will be evaluated, namely as $((a/(b \wedge c)) + (d * e)) - (a * c)$. How can a compiler accept such an expression and produce correct code? The answer is given by reworking the expression into a form we call postfix notation. If e is an expression with operators and operands, the conventional way of writing e is called infix, because the operators come in-between the operands. (Unary operators precede their operand.) The postfix form of an expression calls for each operator to appear after its operands. For example,
infix expression $a * b/c$ has the postfix form : $ab * c/$.

If we study the postfix form of $a * b/c$ we see that the multiplication comes immediately after its two operands a and b. Now imagine that $a * b$ is computed and stored in T. Then we have the division operator, /, coming immediately after its two arguments T and c.

Before attempting an algorithm to translate expressions from infix to postfix notation, let us make some observations regarding the virtues of postfix notation that enable easy evaluation of expressions. To begin with, the need for parentheses is eliminated. Secondly, the priority of the operators is no longer relevant. The expression may be evaluated by making a left to right scan, stacking operands, and evaluating operators using the correct number of operands from the stack and finally placing the result onto the stack. This evaluation process is much simpler than attempting a direct evaluation from infix notation.

Algorithm 7: EVAL(E)

```
   // evaluate the postfix expression E. It is assumed that the last character
      in E is an ∞.  A procedure NEXT-TOKEN is used to extract from E the
      next token.  A token is either a operand, operator, or ∞.  A one
      dimensional array STACK(1 : n) is used as a stack
1  top = 0;
2  while not at end of input do
3      X=NEXT-TOKEN(E);
4      if X=∞ then
5          return // answer is at top of stack
6      if X is an operand then
7          push(X);
8      else
9          remove the correct number of operands for operator X from STACK, perform
             the operation and store the result, if any, onto the stack.
```

To see how to devise an algorithm for translating from infix to postfix, note that the order of the operands in both forms is the same. In fact, it is simple to describe an algorithm for producing postfix from infix:

1. fully parenthesize the expression;
2. move all operators so that they replace their corresponding right parentheses;
3. delete all parentheses.

For example, $a/b \wedge c + d * e - a * c$ when fully parenthesized yields The arrows

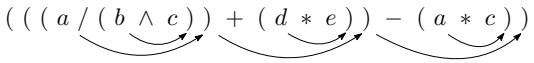


Figure 5: Parenthesized Expression.

point from an operator to its corresponding right parenthesis. Performing steps 2 and 3 gives

$$abc \wedge /de * + ac * -$$

The problem with this as an algorithm is that it requires two passes: the first one reads the expression and parenthesizes it while the second actually moves the operators. As we have already observed, the order of the operands is the same in infix and postfix. So as we scan an expression for the first time, we can form the postfix by immediately passing any operands to the output. Then it is just a matter of handling the operators. The solution is to store them in a stack until just the right moment and then to unstack and pass them to the output. Let us discuss this with few examples. Consider the infix expression $a + b * c$ to yield $abc * +$ in postfix notation. The algorithm should perform following stacking operation (The stack is assumed to grow to the right). The

Sl No	***Next Token***	***Stack***	***Output***
1	none	empty	none
2	a	empty	a
3	$+$	$+$	a
4	b	$+$	ab
5	$*$	$+ *$	ab
6	c	$+ *$	abc

algorithm scans the infix expression from left to right taking one token (operators / operands / left parentheses / right parentheses) until it reaches the end of the input expression. Initially when it encounters 'a', it passes it to the output. The next symbol after 'a' in the infix expression is '+' so it pushes it on to the stack. Next it sees a 'b' which is an operand so passes it to the output. Next to 'b' is an operator '*'. At this point the algorithm must decide if '*' should be pushed on top of the stack or if '+' needs to be popped. Since '*' has higher priority the '+' the algorithm should push '*' on to the stack producing '+ *' on the stack ('*' being at the top). The next symbol after '*' is 'c' and is passed to the output. Now the input expression is exhausted, so the algorithm pops all

remaining operators in the stack to get

$$abc*+$$

Let us now consider another expression, $a*(b+c)*d$ which has the postfix form $abc+*d*$, so the algorithm should behave as follows At this point the algorithm

Sl No	***Next Token***	***Stack***	***Output***
1	none	empty	none
2	a	empty	a
3	*	*	a
4	(	* (	a
5	b	* (	ab
6	+	* (+	ab
6	c	* (+	abc

pops operators from the stack until the corresponding left parenthesis is popped, and then delete the left and right parentheses; this gives us:

7	)	*	$abc+$
8	*	*	$abc+*$
9	d	*	$abc+*d$
10	end of input	empty	$abc+*d*$

It is to be observed that operators are popped out of the stack as long as the priority of the top operator in the stack is greater than or equal to the priority of the new operator. The problem with this rule is that it cannot handle operators which are right-to-left associative. For example consider the expression $a \wedge b \wedge c$, this is same as $(a \wedge (b \wedge c))$ and it should produce $abc \wedge \wedge$ as corresponding postfix expression. If we follow the procedure just mentioned above then it will produce $ab \wedge c \wedge$ as the equivalent postfix expression which is not correct. So, in oder to capture both associativity and precedence of operators following hierarchy scheme (Table. 1) for binary operators and delimiters.

Symbol	In-Stack Priority	In-Coming Priority
)	none	none
$\wedge$	3	4
*, /	2	2
+, -	1	1
(	0	4

Table 1: In-Stack Priority and In-Coming Priority

The overall algorithm for converting an infix expression E into equivalent postfix expression is outlined in the Alg. 8. ISP(x) and ICP(x) are functions which reflect the table . 1

Algorithm 8: PostFix(E)

```
   // convert the infix expression E to postfix.  Assume the last character of
      E is a '∞'.  STACK(1 : n) is used as a stack and the character '∞' with
      ISP('∞') = −1 is used at the bottom of the stack
 1 STACK(1) = ∞;
 2 top = 1;
 3 while not at end of input do
 4 |   x =Next-Token(E)// returns next symbol
 5 |   if x = ∞ then
 6 |   |   while top > 1 do
 7 |   |   |   y = pop();
 8 |   |   |   print(y);
 9 |   |   return;
10 |   if x is an operand then
11 |   |   print(x);
12 |   else if x =') ' then
13 |   |   while (y =pop()) ≠'(' do
14 |   |   |   print(y);
15 |   else
16 |   |   while  ISP(y = pop()) ≥ ICP(x) do
17 |   |   |   print(y);
18 |   |   push(x);
```

4 Multiple Stack and Queues

So far we were concerned only with the representation of a single stack or a single queue in the memory of a computer and we have seen efficient sequential data representations for them. Naturally the question arises, what happens when a data representation is needed to maintain several stacks and queues? Let us once again limit ourselves, to sequential mappings of these data objects into an array, say, $V(1:m)$. If we need to represent only 2 stacks, then the solution is simple. We can use $V(1)$ for the bottom most element in stack 1 and $V(m)$ for the corresponding element in stack 2 with stack 1 growing towards $V(m)$ and stack 2 towards $V(1)$. This solution allows us to utilize efficiently all the available space. Can we do the same when more than 2 stacks needs to be represented? Apparently the answer turns out to be no, because to represent a stack we need a fixed point for its bottommost element and a one dimensional array has only two fixed points $V(1)$ and $V(m)$. To represent more than two, say n, stacks sequentially, we can initially divide out the available memory $V(1:m)$ into n segments and allocate one of these segments to each of the n stacks. This initial division of $V(1:m)$ into segments may be done in proportion to expected sizes of the various stacks if the sizes are known a priory. In the absence of such information, $V(1:m)$ may be divided into equal segments. For each stack i, we shall use $B(i)$ to represent a position one less than the position in V for the bottommost element of that stack. $T(i)$, $1 \leq i \leq n$ will point to the topmost element of stack i. We shall use the boundary condition $B(i) = T(i)$ iff the i'th stack is empty. If we grow the i'th stack in lower memory indexes than the $i+1$'st, then with roughly equal initial segments we have
$B(i) = T(i) = \lfloor \frac{m}{n} \rfloor (i-1)$ for $1 \leq i \leq n$ as the initial values of $B(i)$ and $T(i)$, (see figure 6). Stack i, $1 \leq i \leq n$ can grow from $B(i)+1$ up to $B(i+1)$ before it catches up with the $i+1$'st stack. It is convenient both for the discussion

Algorithm 9: PUSH(i, x)

```
   // Push element x to the
      i'th stack, 1 ≤ i ≤ n
1  if T(i) = B(i + 1) then
2      printf("i'th stack full");
3  else
4      T(i) = T(i) + 1;
5      T(i) = x;
```

Algorithm 10: POP(i)

```
   // Pops topmost element of
      stack i, 1 ≤ i ≤ n
1  if T(i) = B(i) then
2      printf("i'th stack empty");
3  else
4      x = T(i) ;
5      T(i) = T(i) + 1;
6      return x;
```

and the algorithms to define $B(n+1) = m$. Using this scheme thePUSH() and POP() algorithms become:

The algorithms to PUSH() and POP() appear to be a simple as in the case of only 1 or 2 stacks. This really is not the case since the "stack full" condition in algorithm PUSH() does not imply that all m locations of V are in use. In fact, there may be a lot of unused space between stacks j and $j+1$ for $1 \leq j \leq n$ and $j \neq i$. The procedure "stack full (i)" should therefore determine whether there is any free space in V and shift stacks around so as to make some of this free space available to the i'th stack. Several strategies are possible for the design of algorithm "stack full'. We shall discuss one strategy here. The primary objective of algorithm "stack full' is to permit the adding of elements to stacks so long as there is some free space in V. One way to guarantee this is to design "stack full' along the following lines:

1. determine the least j, $i < j \leq n$ such that there is free space between stacks j and $j+1$, i.e., $T(j) < B(j+1)$. If there is such a j, then move stacks $i+1, i+2, \ldots, j$ one position to the right (treating $V(1)$ as leftmost and $V(m)$ as rightmost), thereby creating a space between stacks i and $i+1$.

2. if there is no j as in (1), then look to the left of stack i. Find the largest j such that $1\ j < i$ and there is space between stacks j and $j+1$, i.e.,$T(j) < B(j+1)$. If there is such a j, then move stacks $j+1, j+2, \ldots, i$ one space left creating a free space between stacks i and $i+1$.

3. if there is no j satisfying either the conditions of (1) or (2), then all m spaces of V are utilized and there is no free space.

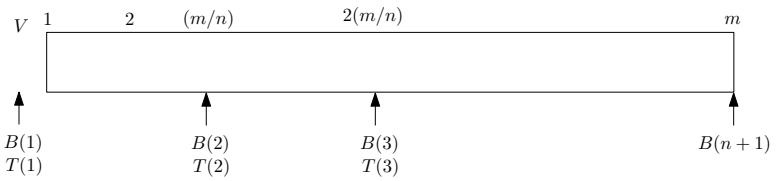


Figure 6: Initial configuration for n stacks in $V(1:m)$.