UNIT – 3 : STACKS AND QUEUES

3.1.Stacks:

A ***stack*** is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the ***top*** of the stack. A stack is a dynamic, constantly changing object as the definition of the stack provides for the insertion and deletion of items. It has single end of the stack as top of the stack, where both insertion and deletion of the elements takes place. The last element inserted into the stack is the first element deleted-***last in first out list (LIFO).*** After several insertions and deletions, it is possible to have the same frame again.

Primitive Operations

When an item is added to a stack, it is ***push***ed onto the stack. When an item is removed, it is ***pop***ped from the stack.

Given a stack *s*, and an item *i*, performing the operation *push(s,i)* adds an item *i* to the top of stack *s.*

push(s, H);

push(s, I);

push(s, J);

Operation *pop(s)* removes the top element. That is, if *i=pop(s)*, then the removed element is assigned to *i.*

pop(s);

Because of the push operation which adds elements to a stack, a stack is sometimes called a ***pushdown list.*** Conceptually, there is no upper limit on the number of items that may be kept in a stack. If a stack contains a single item and the stack is popped, the resulting stack contains no items and is called the ***empty stack.*** Push operation is applicable to any stack. Pop operation cannot be applied to the empty stack. If so, *underflow* happens. A Boolean operation *empty(s)*, returns TRUE if stack is empty. Otherwise FALSE, if stack is not empty.

Representing stacks in C

Before programming a problem solution that uses a stack, we must decide how to represent the stack in a programming language. It is an ordered collection of items. In C, we have ARRAY as an ordered collection of items. But a stack and an array are two different things. The number of elements in an array is fixed. A stack is a dynamic object whose size is constantly changing. So, an array can be declared large enough for the maximum size of the stack. A stack in C is declared as a structure containing two objects:

• An array to hold the elements of the stack.

• An integer to indicate the position of the current stack top within the array.

```
#define STACKSIZE 100

struct stack {

int top;
```

Eduoncloud.com

```
int items[STACKSIZE];

};
```

The stack *s* may be declared by

```
struct stack s;
```

The stack items may be int, float, char, etc. The empty stack contains no elements and can therefore be indicated by *top*= -1. To initialize a stack S to the empty state, we may initially execute

s.top= -1.

To determine stack empty condition,

```
if (s.top=-1)

stack empty;

else

stack is not empty;
```

The empty(s) may be considered as follows**:**

```
int empty(struct stack *ps)

{

if(ps->top== -1)

return(TRUE);

else

return(FALSE);

}
```

Aggregating the set of implementation-dependent trouble spots into small, easily identifiable units is an important method of making a program more understandable and modifiable. This concept is known as ***modularization***, in which individual functions are isolated into low-level modules whose properties are easily verifiable. These low-level ***modules*** can then be used by more complex routines, which do not have to concern themselves with the details of the low-level modules but only with their function. The complex routines may themselves then be viewed as modules by still higher-level routines that use them independently of their internal details.

• **Implementing pop operation**

If the stack is empty, print a warning message and halt execution. Remove the top element from the stack. Return this element to the calling program

```
int pop(struct stack *ps)

{
```


Eduoncloud.com

```
if(empty(ps)){

printf(―%‖,‖stack underflow‖);

exit(1);

}

return(ps->items[ps->top--]);

}
```

3.2. Stacks Using Dynamic Arrays:

For example:

```
 Typedef struct


        {
     char         *str;
        }         words;

        main()
        {
      words x[100]; // I do not want to use this, I want to dynamic increase the size of the array as data
comesin.
   }
```

For example here is the following array in which i read individual words from a .txt file and save them word by word in the array:

Code:

```
char words[1000][15];
```

Here 1000 defines the number of words the array can save and each word may comprise of not more than 15 characters. Now I want that that program should dynamically allocate the memory for the number of words it counts. For example, a .txt file may contain words greater that 1000. Now I want that the program should count the number of words and allocate the memory accordingly. Since we cannot use a variable in place of [1000], I am completely blank at how to implement my logic. Please help me in this regard.

3.3. Queues:

A queue is like a line of people waiting for a bank teller. The queue has a **front** and a **rear**.

When we talk of queues we talk about two distinct ends: the front and the rear. Additions to the queue take place at the rear. Deletions are made from the front. So, if a job is submitted for execution, it joins at the rear of the job queue. The job at the front of the queue is the next one to be executed

• New people must enter the queue at the rear. **push**, although it is usually called an **enqueue** operation.

Edutoncloud.com

• When an item is taken from the queue, it always comes from the front. **pop**, although it is usually called a **dequeue** operation.

What is Queue?

• Ordered collection of elements that has two ends as front and rear.

• Delete from front end

• Insert from rear end

• A queue can be implemented with an array, as shown here. For example, this queue contains the integers 4 (at the front), 8 and 6 (at the rear).

Queue Operations

• Queue Overflow

• Insertion of the element into the queue

• Queue underflow

• Deletion of the element from the queue

• Display of the queue

```
struct Queue {
int que [size];
int front;
int rear;
}Q;
```

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 5
struct queue {
int que[size];
int front, rear;
} Q;
```

Eduoncloud.com

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 5
struct queue {
int que[size];
int front, rear;
} Q;
int Qfull ( ){
if (Q.rear >= size-1)
return 1;
else
return 0;
}
int Qempty( ){
if ((Q.front == -1)||(Q.front > Q.rear))
return 1;
else
return 0;
}
int insert (int item) {
if (Q.front == -1)
Q.front++;
Q.que[++Q.rear] = item;
return Q.rear;
}
Int delete () {
Int item;
```

Eduoncloud.com

```
Item = Q.que[Q.front];
Q.front++;
Return Q.front;
}
Void display () {
Int I;
For (i=Q.front;i<=Q.rear;i++)
Printf(— %d‖,Q.que[i]);
}
Void main (void) {
Int choice, item;
Q.front = -1; Q.Rear = -1;
do {
Printf(—Enter your choice : 1:I, 2:D, 3:Display‖);
Scanf(—%d‖, &choice);
Switch(choice){
Case 1: if(Qfull()) printf(—Cannt Insert‖);
else scanf(—%d‖,item); insert(item); break;
Case 2: if(Qempty()) printf(—Underflow‖);
else delete(); break;
}
}
}
```

3.4. Circular Queues Using Dynamic Arrays:

Circular Queue

• When an element moves past the end of a circular array, it wraps around to the beginning.

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 entered at $Q(0)$ in case that spot is free. Using the same conventions as before, front

will always point one position counterclockwise from the first element in the queue. Again, *front* = *rear* if and only if the queue is empty. Initially we have *front* = *rear* = 1. Figure 3.4 illustrates some of the possible configurations for a circular queue containing the four elements *J*1-*J*4 with *n* > 4. The assumption of circularity changes the ADD and DELETE algorithms slightly. In order to add an element, it will be necessary to move *rear* one position clockwise, i.e.,

if *rear* = *n* - 1 **then** *rear* 0
else *rear* *rear* + 1.

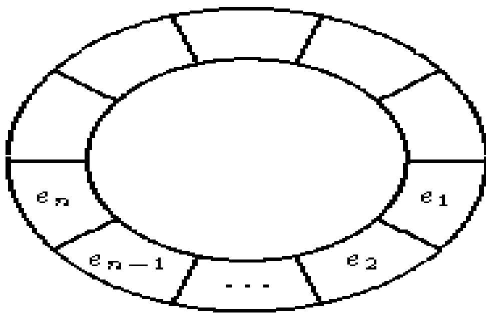

Figure : Circular queue of n elements

Using the modulo operator which computes remainders, this is just *rear (*rear + *1)**mod*** n. *Similarly, it will be necessary to move* front *one position clockwise each time a deletion is made. Again, using the modulo operation, this can be accomplished by* front *(*front + *l)**mod*** n. *An examination of the algorithms indicates that addition and deletion can now be carried out in a fixed amount of time or* O*(1).*

e.g.

• OOOOO7963 _ 4OOOO7963 (after Enqueue(4))

• After Enqueue(4), rear index moves from 3 to 4.

Queue Full Condition:

if(front == (rear+1)%size) Queue is Full

• Where do we insert:

rear = (rear + 1)%size; queue[rear]=item;

After deletion : front = (front+1)%size;

Example of a Circular Queue

• A Circular Q, the size of which is 5 has three elements 20, 40, and 60 where front is 0

and rear is 2. What are the values of after each of these operations:

Q = 20, 40, 60, - , - front–20[0], rear–60[2]

Insert item 50:

Eduoncloud.com

Q = 20, 40, 60, 50, - front-20[0], rear-50[3]

Insert item 10:

Q = 20, 40, 60, 50, 10 front-20[0], rear-10[4]

Q = 20, 40, 60, 50, 10 front-20[0], rear-10[4]

Insert 30

Rear = (rear + 1)%size = (4+1)%5 = 0, hence overflow.

Delete an item

delete 20, front = (front+1)%size = (0+1)%5=1

Delete an item

delete 40, front = (front+1)%size = (1+1)&5=2

Insert 30 at position 0

Rear = (rear + 1)%size = (4+1)%5 = 0

Similarly Insert 80 at position 1

3.5. Evaluation of Expressions: Evaluating a postfix expression:

When pioneering computer scientists conceived the idea of higher level programming languages, they were faced with many technical hurdles. One of the biggest was the question of how to generate machine language instructions which would properly evaluate any arithmetic expression. A complex assignment statement such as

*X A/B ** C + D* E - A * C* (3.1)

might have several meanings; and even if it were uniquely defined, say by a full use of parentheses, it still seemed a formidable task to generate a correct and reasonable instruction sequence. Fortunately the solution we have today is both elegant and simple. Moreover, it is so simple that this aspect of compiler writing is really one of the more minor issues.

An expression is made up of operands, operators and delimiters. The expression above has five operands: *A*,*B*,*C*,*D*, and *E*. Though these are all one letter variables, operands can be any legal variable name or constant in our programming language. In any expression the values that variables take must be consistent with the operations performed on them. These operations are described by the operators. In most programming languages there are several kinds of operators which correspond to the different kinds of data a variable can hold. First, there are the basic arithmetic operators: plus, minus, times, divide, and exponentiation (+,-,*,/,**). Other arithmetic operators include unary plus, unary minus and **mod, ceil**, and **floor**. The latter three may sometimes be library subroutines rather than predefined operators. A second class are the relational operators: . These are usually defined to work for arithmetic operands, but they can just as easily work for character string data. ('CAT' is less than 'DOG' since it precedes 'DOG' in alphabetical order.) The result of an expression which contains relational operators is one of the two

constants: **true** or **false**. Such all expression is called Boolean, named after the mathematician George Boole, the father of symbolic logic.

The first problem with understanding the meaning of an expression is to decide in what order the operations are carried out. This means that every language must uniquely define such an order. For instance, if $A = 4$, $B = C = 2$, $D = E = 3$, then in eq. 3.1 we might want X to be assigned the value

4/(2 ** 2) + (3 * 3) - (4 * 2)
= (4/4) + 9 - 8
= 2.
Let us now consider an example. Suppose that we are asked to evaluate the following postfix expression:

6 2 3 + - 3 8 2 / + * 2 $ 3 +

Symb Opnd1 Opnd2 Value opndstk

6 6

2 6,2

3 6,2,3

+ 2 3 5 6,5

- 6 5 1 1

3 6 5 1 1,3

8 6 5 1 1,3,8

2 6 5 1 1,3,8,2

/ 8 2 4 1,3,4

8

+ 3 4 7 1,7

* 1 7 7 7

2 1 7 7 7,2

$ 7 2 49 49

3 7 2 49 49,3

+ 49 3 52 **52**

Each time we read an operand, we push it onto a stack. When we reach an operator, its operands will be the top two elements on the stack. We can then pop these two elements, perform the indicated operation on them, and push the result on the stack so that it will be available for use as an operand of the next operator. The maximum size of the stack is the number of operands that appear in the input expression. But usually, the actual size of the stack needed is less than maximum, as operator pops the top two operands.

Program to evaluate postfix expression

Along with push, pop, empty operations, we have ***eval, isdigit*** and ***oper*** operations.

eval – the evaluation algorithm

```
double eval(char expr[])
{
int c, position;
double opnd1, opnd2, value;
struct stack opndstk;
opndstk.top=-1;
for (position=0 ;( c=expr [position])!=‘\0‘; position++)
if (isdigit)
push (&opndstk, (double) (c-‗0‘));
else{
opnd2=pop (&opndstk);
9
opnd1=pop (&opndstk);
value=oper(c, opnd1,opnd2);
push (&opndstk. value);
}
return(pop(&opndstk));
}
```

isdigit – called by eval, to determine whether or not its argument is an operand

```
int isdigit(char symb)
{
return(symb>=‘0‘ && symb<=‘9‘);
}
```

oper – to implement the operation corresponding to an operator symbol

```
double oper(int symb, double op1, double op2)
{
```

```
switch (symb){
case ‗+‘ : return (op1+op2);
case ‗-‗ : return (op1-op2);
case ‗*‘ : return (op1*op2);
case ‗/‘ : return(op1/op2);
case ‗$‘ : return (pow (op1, op2);
default: printf (―%s‖,‖illegal operation‖);
exit(1);
}
}
```

Converting an expression from infix to postfix

Consider the given parentheses free infix expression:

A + B * C

Symb Postfix string opstk

1 A A

2 + A +

3 B AB +

4 * AB + *

5 C ABC + *

6 ABC * +

7 ABC * +

Consider the given parentheses infix expression:

(A+B)*C

Symb Postfix string Opstk

1 ((

2 A A (

3 + A (+

4 B AB (+

5) AB+

6 * AB+ *

7 C AB+C *

8 AB+C*

Program to convert an expression from infix to postfix

Along with pop, push, empty, popandtest, we also make use of additional functions such as, *isoperand, prcd, postfix.*

isoperand – returns TRUE if its argument is an operand and FALSE otherwise

prcd – accepts two operator symbols as arguments and returns TRUE if the first has precedence over the second when it appears to the left of the second in an infix string and FALSE otherwise

postfix – prints the postfix string

3.6. Multiple Stacks and Queues:

Up to now we have been concerned only with the representation of a single stack or a single queue in the memory of a computer. For these two cases we have seen efficient sequential data representations. What happens when a data representation is needed for several stacks and queues? Let us once again limit ourselves, to sequential mappings of these data objects into an array $V(1{:}m)$. If we have only 2 stacks to represent. 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. Stack 1 can grow towards $V(m)$ and stack 2 towards $V(1)$. It is therefore possible to utilize efficiently all the available space. Can we do the same when more than 2 stacks are to be represented? The answer is no, because a one dimensional array has only two fixed points $V(1)$ and $V(m)$ and each stack requires a fixed point for its bottommost element. When more than two stacks, say n, are to be represented 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. 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 i 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

$B(i) = T(i) = m/n\ (i - 1)$, 1 i n (3.2)

as the initial values of $B(i)$ and $T(i)$, (see figure 3.9). Stack i, 1 i 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 and the algorithms to define $B(n + 1) = m$. Using this scheme the add and delete algorithms become:

procedure *ADD*(*i*,*X*)
//add element *X* to the *i*'th stack, 1 i *n*//
if *T*(*i*) = *B*(*i* + 1) **then call** *STACK-FULL* (*i*)
T(*i*) *T*(*i*) + 1
V(*T*(*i*)) *X* //add *X* to the *i*'th stack//
end *ADD*
procedure *DELETE*(*i*,*X*)

//delete topmost element of stack *i*//
if *T*(*i*) = *B*(*i*) **then call** *STACK-EMPTY*(*i*)
X V*(*T*(*i*))*
T(*i*) *T*(*i*) - 1
end *DELETE*

The algorithms to add and delete 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 ADD 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 *j n* and *j 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 in the text and look at some others in the exercises. 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:

a) determine the least *j, i* < *j 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, ...,*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.

b) if there is no *j* as in a), 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, ...,*i* one space left creating a free space between stacks *i* and *i* + 1.

c) if there is no *j* satisfying either the conditions of a) or b), then all *m* spaces of *V* are utilized and there is no free space.

It should be clear that the worst case performance of this representation for the *n* stacks together with the above strategy for STACK_FULL would be rather poor. In fact, in the worst case *O*(*m*) time may be needed for each insertion (see exercises). In the next chapter we shall see that if we do not limit ourselves to sequential mappings of data objects into arrays, then we can obtain a data representation for *m* stacks that has a much better worst case performance than the representation described here.

3.7. RECOMMENDED QUESTIONS

1. Assume A=1, B=2, C=3. Evaluate the following postfix expressions : a) AB+C-BA+C$-
   b) ABC+*CBA-+*

2. Convert the following infix expression to postfix : ((A-(B+C))*D)$(E+F)

3. Explain the different ways of representing expressions

4. State the advantages of using infix & postfix notations

5. State the rules to be followed during infix to postfix conversions

6. State the rules to be followed during infix to prefix conversions

7. Mention the advantages of representing stacks using linked lists than arrays