time, but a good executive would presumably manage to get the stack emptied periodically. It turns out that sometimes a computer program is naturally organised in this way, postponing some tasks and doing others. Thus, pushdown stacks appear as the fundamental data structure for many algorithms. We may draw a stack in any one of the forms as in Fig. 1. Each one of the

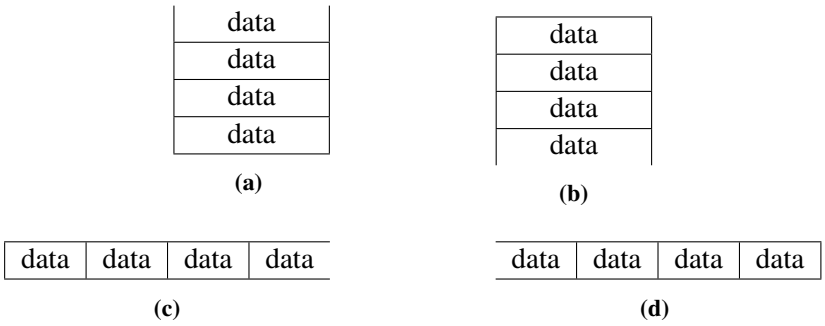


Fig. 1: Depicting stacks.

above have an open and one closed end. The data movement(i.e. storage and retrieval) takes place only at the open end, i.e. data is stored and retrieved in last in first out(LIFO) order. We generally use the form given in 1a, having the open end in up direction. We will see a great many applications of stack in the unit.

The open end of the stack is called the top of the stack. The store and retrieval operations for a stack are called **PUSH** and **POP**, respectively. Fig. 2 shows how a sample stack evolves through the series of PUSH and POP represented by the sequence:

A*SAM*P*L*ES*T***A*CK**

Each letter in this list means “PUSH”(the letter); each asterisk means “POP”. A “PUSH” operation on an object places it on the top of the stack while a “POP” operation removes the top most object on the stack for carrying out some operation or other.

In Fig. 2, there are 16 columns, including the first column as the first column, which contains the labels of the 4 rows. In first row we have a symbol, a character or an asterisk. In the next 2 rows, we have the action initiated by the symbol, either a PUSH or a POP operation. In the fourth row, we have the status of the stack after the operation. For example, in the first row the

Symbol	A	*	S	A	M	P	L	E	*	*	*	*	*	*	*
PUSH	A		S	A	M	P	L	E							
POP		A							E	L	P	M	A	S	
Stack	A		S	A	M	P	P	E	L	P	M	A	S		
				S	A	M	M	L	P	M	A	S			
					S	A	A	P	M	A	S				
						S	S	M	A	S					
								A	S						
								S							

Fig. 2: An illustration of stack operations.

second column, we have ‘A’. Since this pushed A into the stack, we have the character ‘A’ in the second row. Nothing has been popped, so the third row is empty. The last row shows the stack with the character ‘A’. In the third column, we have the asterisk in the first row, which indicates a POP operation. The row corresponding to POP operation has ‘A’, since ‘A’ has been popped. In this last row we have an empty stack since we have popped the only character in the stack. Now you can study the remaining columns to understand the PUSH and POP operations.

Another fundamental restricted-access data structure is called the Queue. Again, only two basic operations are involved; one can insert an item into the Queue at the beginning and remove an

item from the end. Perhaps our busy executive's 'in' box should operate like a Queue, since then work that arrives first would get done first. In a stack, something can get buried at the bottom, but in a Queue everything is processed in the order received. Queues obey a "First In First Out(FIFO)" discipline. We may draw Queue in any one of the forms given in Fig. 3. Queue is marked with two open ends called **front** and **rear**. In the next section, we will discuss

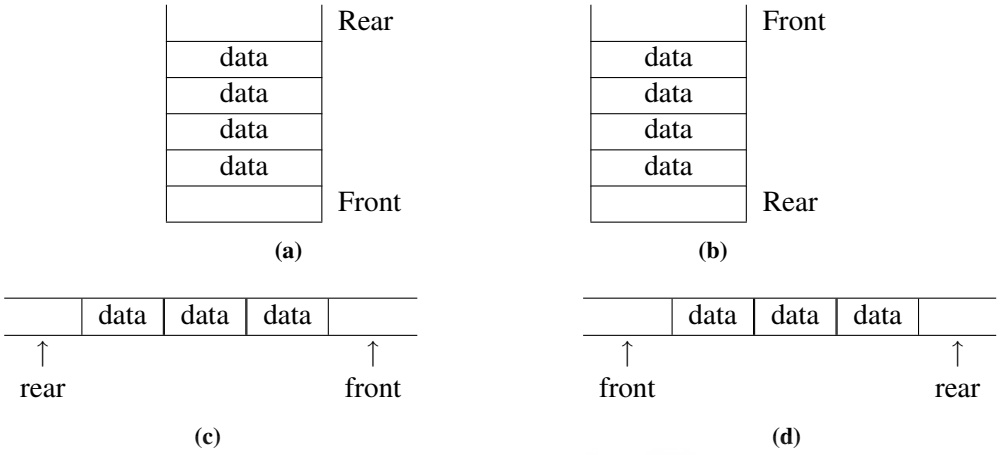


Fig. 3: Depiction of Queue.

implementation of stacks.

13.3 STACK OPERATIONS AND IMPLEMENTATIONS

Basic operations on stack are as follows:

1) Create a stack.
2) Check whether a stack is empty.
3) Check whether a stack is full.
4) Initialise a stack.
5) Push an element onto a stack(if stack is not full).
6) Pop an element from a stack(if stack is not empty).
7) Read a stack top.
8) Print the entire stack.

Stack(a special case of list) can be implemented as one of the following data structures:

- Array
- Linked list

13.3.1 Array Implementation

The simplest way to represent a stack is by using a one-dimensional array, say `stack[N]` with room for N elements. The first element will be `stack[1]`, and so on. An associated variable `top` points to the top element of the stack. Type definition for a sequentially allocated stack is

```
#define STACK_SIZE_MAX 100 /* Maximum stack size.*/
typedef struct {int key;
}element; element
stack[STACK_SIZE_MAX];
int top = -1;/*Denotes an empty stack.*/
```

To check whether the stack is empty, we just need to check the value of `top`. If it is empty, the value of `top` is -1 and the function returns 0; otherwise it returns 1.

```
int stackempty()
{
    if (top == -1)
        return 1;
    else
        return 0;
}
```

Given a sequentially allocated stack, and a value to be pushed, this procedure makes the new top of the stack to be that value.

```
void add (int *top, element item)
{
    if (*top >= STACK_SIZE_MAX - 1) {
        Error("Stack overflow!");
        return;
    }
    stack[++*top]=item;
}
```

E1) Write a function POP to pop a stack.

13.3.2 Pointer Implementation

Although this method of allocating storage is adequate for many applications, there are many other applications where the sequential allocation method is inefficient and therefore not acceptable.

For such applications, we store a stack element in a structure with a pointer to the next lower element on the stack.

```
typedef struct node {
    int data;
    struct node *next;
} Item;
```

Suppose S is a pointer to topmost node in the stack. Here is a function to check if the stack is empty.

```
int IsEmpty(Item * S)
{
    if (S != NULL)
        return 0;
    else
        return 1;
}
```

PUSH, POP and TOP operations involve inserting, deleting and reading item at the top of this list structure.

Here is how we carry out the Pop operation.

```
void Pop(Item ** S)
{
    Item *current, *FirstCell;
```

```
current = *S;
if (IsEmpty(*S)) {
    printf("Empty Stack");
    return;
} else {
    FirstCell = current;
    *S = current->next;
    free(FirstCell);
}
```

We leave the rest of the operations as exercises to you.

E2) Write C functions for carrying Push and Top operations. Also, write a function that prints the contents of the stack. Write a small C function that

a) Pushes 4, 5 and 7 into the stack.

b) Prints the contents of the stack.

c) Pops the stack.

d) Prints the contents of the stack again.

In the next section, we will see some applications of stacks.

13.4 STACK APPLICATIONS

Stacks are simple structures that figure prominently in many algorithms.

Many algorithms implement basic stack operations in hardware because they naturally implement function call mechanisms: Save the current environment on entry to a procedure by pushing information onto a stack, restore the environment on exit by using information popped from the stack. Some calculators and some computing languages base their method of calculations on stack operations explicitly: Every operation pops its arguments from the stack and returns its results to the stack. In this section, we shall consider two of the many applications of stacks.

13.4.1 Infix to Postfix Conversion

You are already familiar with infix notation where the operator is placed between the operands as in the expression $2+3$. Here the operator $+$ is placed between the operands 2 and 3. When we have an expression like $2+3\cdot 5$, we first evaluate $3\times 5=15$ and add it to 2, because multiplication and division have higher priority than addition and subtraction. If we want to change priorities we use brackets; in $(2+3)\cdot 5$, we first add 2 and 3 and add multiply the result by 5.

Polish logician Łukasiewicz invented a postfix notation for writing expressions without brackets in 1920s. In a postfix notation, we write the operator *after* the operands as in $2,3,+$.(Usually the operators are separated by spaces in postfix notation. Here, we are using comma to increase clarity.) The Reverse Polish Notation was invented by Charles Hamblin in mid 1950s. We can convert any notation in infix notation to this notation in a unambiguous way. Converting an expression in infix notation to RPN is an interesting application of stacks.

Before we proceed further, let us see how we can evaluate an expression in RPN. Consider the following expression in RPN:

$$3,5,-2,3,+,/,5,+ \qquad (1)$$

The rule is

1. We read the expression from left to write till we reach an operator.
2. Apply the operator to the two operands immediately preceding the operator.
3. Replace the operator and the two operands by the answer and continue reading right.

When we apply this rule, this is how we will evaluate the expression in eqn. (1) on the preceding page. The details are in Table 1.

Table 1: Evaluation of an expression in RPN notation.

$\boxed{3,5,-}\,2,3,+,/,5,+$	The first operator we encounter is $-$. Apply it to the two operands 3 and 5. Replace 3, 5 and $-$ by -2 in the expression.
$-2,\boxed{2,3,+}\,,/,5,+$	The next operator is $+$. Apply it to 2 and 3 and replace 2, 3 and $+$ by 5 in the expression.
$\boxed{-2,5,/}\,,5,+$	The next operator is /. Replace -2, 5 and / by $-\frac{2}{5}$ in the expression.
$\boxed{-\frac{2}{5},5,+}$	The next operator is $+$. Apply it.
$5-\frac{2}{5}=\frac{23}{5}$	There are no operators left. Answer is $\frac{23}{5}$.

How do we convert an expression in infix notation to RPN? Consider the expression $5+(3-5)/(2+3)$. We first convert the expressions in brackets to reverse polish notation. The expression becomes $5+[3,5,-]/[2,3,+]$. We treat the converted forms in square brackets to be operands.

There are no more expressions in round brackets to be converted. Now, we apply the priority rules. / has higher priority than $+$, so the expression becomes $5+[[3,5,-],[2,3,+],/]$. In the next step, we convert this to $[5,[[3,5,-],[2,3,+],/],+]$. Now, we write the expression without square brackets.

$$5,3,5,-,2,3,+,/,+$$

Here is an exercise for you.

E3) Evaluate the expression $5,3,5,-,2,3,+,/,+$ and check that its value is the same as $5+(3-5)/(2+3)$.

We may write a general algorithm as follows:

1) Initialise the stack to be empty.
2) For each character in the input string, if it is an operand, append it to the output. If it is an operator that has higher precedence than the operator on the top of the stack or if the stack is empty, push it onto the top of the stack. If the incoming operator has the same or lower precedence than the operator on the stack, pop the stack and append it to the output. Repeat this process till the operator on the top of the stack has lower precedence than the incoming operator or the stack is empty. After this, add the incoming operand to the top of the stack.
3) If the input end is encountered, pop the elements in the stack one by one and append them to the output.

Let us write a C programme that converts an infix expression to RPN. We will put some restrictions on the infix expression to keep our program simple. We will assume

1. The numbers are single digit numbers and negative integers are not allowed.

2. The expression does not contain round brackets; we will do the conversion purely according to priority. Also, we will assume that the expression does not contain the division operator /.

Here is the program; this uses the stack functions we defined earlier.

```
/*Program-13.2. A program that converts an
infix expression to RPN.
File name:unit13-infix2postfix.c*/
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#define MAX_STRING 20
typedef struct node {
    int data;
    struct node *next;
} Item;
int IsEmpty(Item * S);
void EmptyStack(Item ** S);
void Push(int x, Item ** S);
int Top(Item * S);
void Pop(Item ** S);
void print_stack(Item * S);
void Error(char *message);
int ishigher(char op1, char op2);
int main()
{
    Item *S=NULL;
    char c,op, out[MAX_STRING];
    int i=0,j=0;
    while((c=getchar()) != '\n'){
        if (c == ' ')
            continue;
        if(isdigit(c))
            out[i++]=c;
        else if(IsEmpty(S))
            Push(c,&S);
        else{
            while(!IsEmpty(S) && ishigher(Top(S),c)){
                out[i++]=Top(S);
                Pop(&S);
            }
            Push(c,&S);
        }
    }
    while(!IsEmpty(S)){
        out[i++]=Top(S);
        Pop(&S);
    }
    out[i]='\n';
    for(j = 0; j <= i-1; j++)
        printf("%c",out[j]);
    return(0);
}
```

Listing 1: A program to convert an infix expression to RPN.

We have used the function `isdigit()` from the standard C library which is defined in `ctypes.h` to checks whether the character input is a digit or a non-digit. The only new thing in the program is the function `ishigher()` which checks the priority of the operators and

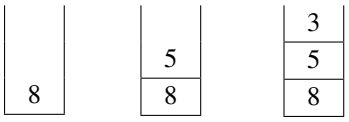
returns 1 or 0 depending on whether the first operator has higher priority than the second operator or not. We ask you to write such a function in the next exercise.

E4) Write a function `ishigher()` that checks the priority of operators as described above.

We can also use stacks to evaluate expressions in postfix notation. To do this, when an operand is encountered, it is pushed onto the stack. When an operator is encountered, it is applied to the first two operands that are obtained by popping the stack and the result is pushed onto the stack. For example, the postfix expression

$$853+9*+4+$$

is evaluated as follows: On reading 8, 5 and 3 the stack contents are as follows:



The remaining steps are shown in Table 2.

Table 2: Evaluation of an expression in RPN using a stack.

Step	Stack
On reading $+$, 3 and 5 are popped from the stack and added. The result $8 = 5 + 3$ is pushed onto stack	8 8
Next, 9 is pushed onto the stack	9 8 8
On reading $*$, 8 and 9 are popped and $9 * 8 = 72$ is pushed onto the stack	72 8
On finding $+$, 72 and 8 are popped out and $72 + 8 = 80$ is pushed onto the stack	 80
Now 4 is pushed onto the stack	4 80
Finally, a $+$ is read and 4 and 80 are popped, the result $4 + 80$ is pushed onto the stack.	 84

End of the string is encountered. Therefore, stack is popped and 84 is is the result.

E5) Write a function that evaluates an expression in RPN that is given as string. You can make all the assumptions that we made in the program we wrote for converting an infix expression to an expression in RPN.

While recursive algorithms are elegant, not all programming languages support recursion. In the next subsection, we will see how to simulate recursion in such languages.

13.4.2 Simulation of Recursion Using Stacks

The discussion in this subsection is based on Section 3.4 of the book *Data Structures using C and C++, Second edition* by *Y. Langsam, M. J. Augenstein and A. M. Tenenbaum*. Recall the factorial function, that can be defined recursively.

```
1  int fact(int n)
2  {
3      int x, y;
4      if (n == 0)
5          return (1);
6      x = n - 1;
7      y = fact(x);
8      return (n * y);
9  }
```

We can write this program in a non-recursive form by simulating this recursive solution using elementary operations. Why should we simulate recursion?

Many programming languages like FORTRAN, COBOL and the machine languages do not support recursion. So, in these cases, we can easily find a recursive solution to a problem and write a non-recursive program for it. Also, sometimes, a recursive solution can be more expensive in terms of time and space. Often the difference is small and can be ignored because recursive solutions are logically simple and elegant. However, in some cases where the program will be run thousands of times, using recurrence may prove to be expensive. However, with increase in computing power and easy availability of cheap memory, such instances are fewer than before.

So, when a recursive solution proves to be costly in terms of running time and memory used, the best alternative for us is to solve a problem using recursion and then convert it into a non-recursive form. Stacks are useful in this regard. As an example of this procedure, we will rewrite the above factorial function in non-recursive form.

For this, you may look at the introduction to Unit 8 to recall how function calls work. Let us see what happens in these steps.

1) **Passing arguments:** For a parameter of the function, its argument is copied within the data area of the function and all the changes are made to the local copy. The original data cannot be altered.
2) **Allocating and initialising local variables:** These local variables include all those declared in the function as well as the **temporaries** that are created during execution. For example, in a statement such as `x=fact(n);` a temporary must be created to hold the value of `fact(n)` before it is assigned to `x`.
3) **Transferring control to the function:** Recall that PC, the Programme Counter holds the address of the next instruction to be carried out. The function has to return control to this address when it has finished its job. So, this address, called the **return address** has to be stored. After the arguments and return address are passed, the program hands over the control to the function.

When the function returns control, first the return address is retrieved and stored in a safe place. After this, the data area of the function, containing all the local variables(including local copies of arguments), temporaries and return address, is freed. The control is returns to the program and it starts execution from the return address. The value returned by the function, if any, is stored away safely.

Suppose the main function calls a function f_1 and f_1 in turn calls another function f_2. The

situation is as shown in Fig. 4. Here the control is somewhere in f_2. Each function has a location

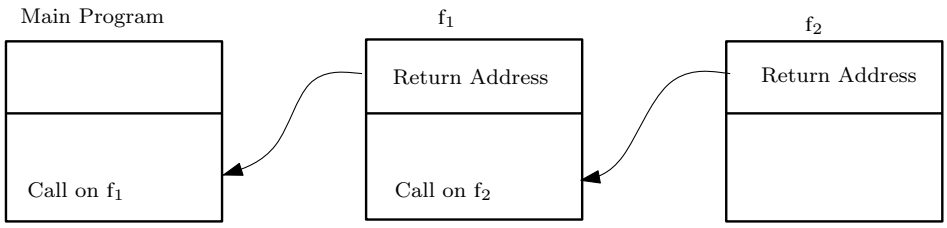


Fig. 4: Series of function calls.

for return address. The return address area of f_2 has the address of the instruction in f_1 immediately following the call to f_2. Similarly, f_1 has the address of the instruction in the main program immediately following the call to f_1. If there are any function calls made by f_2, f_2 returns the control to f_1 after all of them have returned the control back to it. Similarly, f_1 cannot return control to the main program till control returns to it from f_2. So, the series of return addresses have a natural structure of a stack. At any point, we can only access the return address from the function that is currently executing.

We will now use a stack to simulate the recursive calls of `fact`. Each item in the stack will contain the 'return address' of the function call and the temporaries. The item in the stack must contain `x`, `n` and `y`. The control will return to the function at either to the assignment of `fact` to `y` or to the main program `fact(x)`. We will have two labels, `label1` and `label2`. Let `label2` be the label of a section of a code

```
label2: y = result;
```

and let `label1` be the label of the statement

`label1:` **`return`**`(result)` The variable `result` will be used to store the value returned by a call of the `fact` function. We will store the return address in an integer i which will be either 1 or 2. We use the **`switch`**`()` statement

```
switch(i){
    case 1:
        goto label1;
    case 2:
        goto label2;
}
```

for returning from a recursive call. The data area stack for this example is as follows:

```
#define MAXSTACK 50
struct dataarea
{
    int param,
    int x;
    long int y;
    short int retaddr;
};
struct stack{
    int tmp
    struct datararea item[MAXSTACK]
};
```

We use `param` in the data area to store the different parameter values with the simulated function is called. We also declare a current data area to hold the values of the variables in the simulated "current" call on the recursive function. The declaration is:

```
struct dataarea currarea;
```

In addition, we declare a single variable result by

```
long int result;
```

This variable is used to communicate the returned value of the `fact` from one recursive call of fact to the outside calling function. Since the elements on the stack of data areas are structures and it is more efficient to pass structures by reference, we do not use a function `pop` to pop a data area from the `stack`. Instead, we write a function `popsub` defined by

```
void popsub(struct stack *ps, struct datarea *parea)
```

The call `popsub(&s,&area)` pops the stack and sets `area` to the popped element.

A return from fact is simulated by the code.

```
result = value to be returned;
i = currarea.retaddr;
popsub(&s, &currarea);
switch(i){
    case 1: goto label1;
    case 2: goto label2;
}
```

To simulate a recursive call on fact we push the current data area on the stack, reinitialise the variables `currarea.param` and `currarea.retaddr` to the parameter and return address of this call, respectively and then transfer the control to the start of the simulated routine. Note that `currarea.x` holds the value of $n-1$ which will be the new parameter. On recursive call, we wish to eventually return to label2. The code to accomplish this is

```
push(&s, &currarea);
currarea.param= currarea.x;
currarea.retaddr =2;
goto start;
/* start is the label of the  start of
the simulated routine*/
```

Of course, we have to write the pobsub and push routines in such a way that they pop and push entire structures of type dataarea rather than simple variables. Another constraint arising from the array implementation of stacks is that variables `currarea.y` must be initialised to some value or an error will result in the push routine on assignment of currarea.y to the corresponding field the top data area when the program starts.

When the simulation first begins. we initialise the current area so that `currarea.param` equals `n` and `currarea.retaddr` equals 1(indicating a return to the calling routine). We must push a dummy data area onto the stack so that when popsub is executed in returning to the main routine, an underflow does not occur. We should also initialise this dummy data area so as not to cause an error in the push routine. Thus, the simulated version of the recursive fact routine is as follows:

```
struct dataarea{
int param;
int x;
long int y;
short int retaddr;
};
struct stack {
int top;
struct dataarea item[MAXSTACK];
};
int simfact(int n)
{
        struct dataarea currarea;
        struct stack s;
        short int i;
        long int result;
```

```
        s.top = -1;
/*Initialise dummy data area.*/
        currarea.param = 0;
        currarea.x=0;
        currarea.y=0;
        currarea.retaddr=0;
/*Push the dummy data area on the stack*/
        push(&s, &currarea);
/*set the parameter and the return address
        of the current data area to their
        proper values.*/
        currarea.param = n;
        currarea.retaddr=1;
start:/*This is the beginning of the simulated
        factorial routine.*/
        if (currarea.param == 0){
                /*Simulate return(1)*/
                result =1;
                i=currarea.retaddr;
                popsub(&s,&currarea);
                switch(i){
                        case 1: goto label1;
                        case 2: goto label2;
                }/*end switch*/
        }/*end if*/
        currarea.x=currarea.param-1;
        /*simulation of recursived call to fact.*/
        push(&s, &currarea);
        currarea.param = currarea.x;
        currarea.retaddr=2;
        goto start;
label2:/*This is the point to which we return from
        the recursive call. set currarea.y to the returned value.*/
        currarea.y=result;
/*simulation of return(n*y)*/
        result=currarea.param*currarea.y;
        i=currarea.retaddr;
        popsub(&s, &currarea);
        switch(1){
                case 1: goto label1;
                case 2: goto label2;
                }/*end switch*/
label1:/*At this point we return to the main routine.*/
                return(result);
        }/*end simfact*/
```

Work through the program for n = 5 and be sure that you understand what the program does and how it does it. Notice that no space was reserved in the data area for temporaries, since they need not be saved for later use. The temporary location that holds the value of `n*y` in the original recursive routine is simulated by the temporary for `currarea.param=currarea.y` in the simulating routine. This is not the case in general. For example, if a recursive function `funct` contained a statement such as

```
x=a*funct(b)+c*funct(d);
```

we have to save the temporary for `funct(b)` during the recursive call on funct(d). However, this is not required in our example.

So, which temporary variables need to be stacked? We must save a variable on the stack only if the value at the point of initiation of recursive calls must be reused after return from that call. Let us examine whether this is so regarding the variables `n`, `x` and `y`. Clearly, we do not have to stack `n`. In the statement

the old value of n must be used in the multiplication after return from the recursive call on fact. However this is not so for x and y. In fact, the value of y is not even defined at the point of the recursive call, we need not put it on the stack. Similarly, we need not stack x also because we do not use it again after returning.

If we declare x and y as global variables rather than within the recursive function, the routine will work fine. Thus, automatic stacking and unstacking of x and y is unnecessary.

Another issue is whether the return address is really needed in the stack. We make only one textual recursive call to fact and so there is only one return address within fact. But, suppose a dummy data area had not been stacked upon initialising the simulation. Then, a data area is placed on the stack only in simulating a recursive call. When the stack is popped in returning from a recursive call, that area is removed from the stack. However, when we attempt to pop the stack in simulating a return to the main procedure, an underflow will occur. We can test for this underflow by using popandtest rather than popsub and when it does occur we can return directly to the outside calling routine rather than through a local label. This means that one of the return addresses can be eliminated. Since this leaves only a single possible return address, it need not be placed on the stack.

Thus, we have reduced the data area and the stack may be declared by

```
#define MAXSTACK 50
struct stack{
  int top;
  int param[MAXSTACK]; }
```

The current data area is reduced to a single variable declared by

```
int currparam;
```

The program is more comprehensible now:

```
int simfact(int n);
{
        struct stack s;
        short int und;
        long int result,y;
        int currparam,x;

        s.top = -1;
        currparam = n;
start:/*This is the beginning of the simulated
        factorial routine.*/
        if(currparam == 0){
                /* simulation of return(1)*/
                result = 1;
                popandtest(&s, &currparam, &und);
                switch(und) {
                        case FALSE: goto label2;
                        case TRUE: goto label1;
                        }/*End switch*/
                        /*currparam !=0 */
                        x = currparam-1;
                        /*simulation of the recursive call to fact*/
                        push(&s, currparam);
                        currparam = x;
                        goto start;
label2: /*This is the point to which we return from
                        recursive call.  Set y to the returned
                        value.*/
                        y = result;
```

```
                                /*Simulation of return(n*y)*/
                                result = currparam*y;
                                popandtest(&s,&currparam, &und);
                                switch(und){
                                case TRUE: goto label1;
                                case FALSE: goto label2;
                                }/*End Switch*/
label1:/*At this point we return to the main
                                routine*/
                                return(result);
                        }/*end simfact*/
```

Still, the program is far from ideal. The **goto** statements interrupt the flow of thought at a time when one might understand what is happening. Let us see if we can improve the program further.

The statement

```
popandtest(&s,&currparam, &und);
switch(und){
    case TRUE: goto label1;
    case FALSE: goto label2;
}/*End Switch*/
```

is repeated twice for the two cases `currparam == 0` and `currparam != 0`. The two sections can be combined in one.

Also, the two variables `x` and `currparam` are assigned values from each other and are never in use simultaneously. Therefore, we can combine them as one variable and refer to it as `x`. The same is true of `result` and `y`. We combine them into a single variable `y`. After these changes the program becomes:

```
struct stack {
        int top;
        int param[MAXSTACK];
};
int simfact(int n)
{
        struct stack s;
        short int und;
        int x;
        long int y;
        s.top = -1;
        x=n;
start:/*This is the beginning of the simulated
        factorial routine.*/
        if(x == 0)
                y=1
        else {
                push(&s,x--);
                goto start;
        }/*end else*/
label1: popandtest(&s, &x, &und)
        if(und == TRUE)
                return(y);
label2:y *= x;
        goto label1;
}/*end simfact*/
```

There are two loops in the program.

1) The loop that consists of the entire **if** statement, labelled `start`. This loop is exited when `x` equals 0, at which point y is set to 1 and execution proceeds to the label `label1`.

2) The loop that begins at label `label1` and ends with the statement **goto** `label1`. This loop is exited when the stack has been emptied and underflow occurs, at which point a return is executed. These loops can be easily transformed into explicit **while** loops as follows:

```
/*Subtraction loop*/
start: while(x != 0)
           push(&s, x--);
       y = 1;
       popandtest(&s, &x, &und);
label1: while (und == FALSE){
            y *= x;
            popandtest(&s, &x, &und);
        }/*end while*/
        return(y);
```

Let us examine these two loops more closely. `x` starts off at the value of the input parameter `n` and is reduced by 1 each time that the subtraction loop is repeated. Each time `x` is set to a new value, the old value of `x` is saved on the stack. This continues until `x` is 0. Thus, after the first loop has been executed the stack contains, from top to bottom, the integers 1 to n.

The multiplication loop merely removes each of these values from the stack and sets `y` to the product of the popped value and the old value of y. Since we know what the stack contains at the start of the multiplication loop, why bother popping the stack? We can use those values directly. We can eliminate the stack and the first loop entirely and replace the multiplication loop with a loop that multiplies `y` by each of the integer from 1 to n in turn. The resulting program is

```
int simfact(int n)
{
    int x;
    long int y;
    for (y = 1; x <= n; x++)
        y *= x;
    return (y);
}                                /*end simfact */
```

This program is but a direct C implementation using iteration!

13.5 QUEUES: OPERATIONS AND IMPLEMENTATION

In multiuser system, there will be requests from different users for CPU time. The operating system puts them in queue and they are disposed on FIFO(First In, First Out) basic. We will discuss the queue data structure and the operations that it allows in this section. Similar to stack operations, operations that can be carried out on a queue are:

1) Create a queue.
2) Check whether a queue is empty.
3) Check whether a queue is full.
4) Add item at the rear of the queue(enqueue).
5) Remove item from front of queue(dequeue).
6) Read the front of the queue.
7) Print the entire queue.

As we did in the case of stacks, we can give a array representation of a queue. We define a queue as a structure containing the array and two variables, front and rear to denote the present position of its front and rear elements.

We may define a queue as follows:

```
const max = 100;
typedef struct q_type{
    elementtype queue[max];
    int front, rear;
}Qtype;
```

As an example of this representation of a queue, consider a queue of size 6. Assume that queue is initially empty. We want to insert elements RED, BLACK and BLUE, delete RED and BLACK and insert GREEN, WHITE and YELLOW.

Following figure gives a trace of the queue contents for this sequence of operations:

RED					
RED	BLACK				
RED	BLACK	BLUE			
	BLACK	BLUE			
		BLUE			
		BLUE	GREEN		
		BLUE	GREEN	WHITE	
		BLUE	GREEN	WHITE	YELLOW

Now, if we try to insert ORANGE, an overflow occurs even though the first two cells are free. To avoid this drawback, we can arrange these elements in a circular fashion with QUEUE[0] following QUEUE[N-1]. It is then called a circular array representation. We may depict a circular queue as shown in Fig. 5. We also initialise the values of `q.front` and `q.rear` to

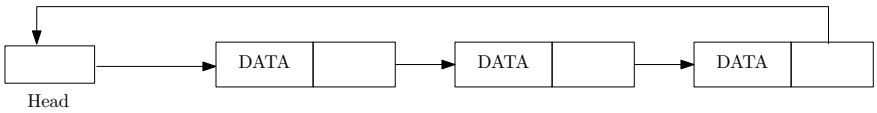


Fig. 5: Circular queue.

`max-1`. So, initially, when the queue is empty, `q->front` and `q->rear` have the same value.

The procdeure for checking whether a circular queue is empty and for inserting and elements in a circular queue are given below:

```
int empty(Qtype *q)
{
    return((q->front == q->rear)?1:0);
}
void qinsert(Qtype *q, int x)
{
    int newrear;
    if (q->rear == max-1)
        newrear = 0;
    else
        newrear = q->rear + 1;
    if (newrear == q->front);
        Error("QUEUE OVERFLOW");
    else{
        q->rear=newrear;
        q->queue[q->rear]=x;
```

```
    }
}
int qdelete(Qtype *q)
{
    if (empty(q)){
        Error("Underflow!");
        exit(1);
    }
    if (q->front == max-1)
        q->front = 0;
    else
       (q->front)++;
    return(q->queue[q->front]);
}
```

Queues are important in simulation models. They serve several purposes like repositories for scheduled events, holding areas for entities moving through the system etc.

Similarly, we may write procedures for other operations on queues.

The second approach for implementing queues is by using the dynamic storage allocation through the use of pointers. We can define a queue consisting of records where each record contains a pointer to the record that comes after it. Therefore, we may declare a queue as

```
struct qrec{
elementtype data;
struct qrec *next;
};
struct qrec *qptr
```

We must also specify the items at the front and rear of the queue. This may be done by the following declaration.

```
struct qtype{
struct qrec *front;
struct qrec *rear;
};
qtype q;
```

This kind of an implementation is a singly linked list implementation of the queue. Recall the Queue implementation using circular arrays. Similarly, we can implement queues using circular lists. In this list, each node points to the next and the chain of pointers eventually form a loop back to the first one.

In such a case the declaration becomes simplified as follows:

```
struct qrec{
elementtype data;
struct qtype next;
};
struct qrec *qtype;
struct qrec *q;
```

You may notice that circular list implementation requires special attention for insertion and deletion with no elements or with just one element.

13.6 PRIORITY QUEUES

Many applications involving queues require priority queues rather than simple FIFO strategy. Each queue element has a associated priority value and the elements are served on a priority basis instead of using the order of arrival. For elements of same priority, the FIFO order is used.

For example, in a multi-user system, there will be several programmes competing for use of the central processor at the same time. The programs have a priority value associated to them and are held in a priority queue. The program with the highest priority is given the first use of the central processor.

Scheduling of jobs within a time-sharing system is another application of queues. In such a system many users may request processing at a time and computer time is divided among these requests. The simplest approach sets up one queue that store all requests for processing. Computer processes the request at the front of the queue and finishes it before starting on the next. Same approach is also used when several users want to use the same output device, say a printer.

In a time sharing system, another common approach used is to process a job only for a specified maximum length of time. If the program is fully processed within that time, then the computer goes on to the next process. If the program is not completely processed within the specified time, the intermediate values are stored and remaining part of the program is put back on the queue. This approach is useful in handling a mix of long and short jobs.

13.7 SUMMARY

A stack is a list in which retrievals, insertion and deletions take place at the same position. It follows the last in first out (LIFO) mechanism. Compiler implements recursion by generating code for creating and maintaining an activation stack, i.e. a run time stack that holds the state of each active subprogram. A queue is a list in which retrievals and deletions can take place at one end and insertions occur at another end. In follows first in, first out(FIFO) order.

Queues are employed in many situations. The items on queues may be vehicles waiting at a crossing, cars waiting at the service station, customers in a cinema ticket counter etc.

13.8 SOLUTIONS/ANSWERS

E1)
```
void POP(int *top)
{
    if(stackempty())
        Error("No element to POP.");
    else
        --*top;
}
```

E2) Function for Push operation

```
void Push(int x, Item ** S)
{
    Item *tmp;
    tmp = malloc(sizeof(Item));
    if (tmp == NULL) {
        Error("Out of space!");
        exit(1);
    } else {
        tmp->data = x;
        tmp->next = *S;
        *S = tmp;
    }
}
```

Function for Top operation.

```
int Top(Item * S)
```

```
{
    if (!IsEmpty(S))
        return S->data;
    printf("Empty Stack!");
    return 0;
}
```

Function for printing the stack.

```
void print_stack(Item * S)
{
    if (IsEmpty(S))
            printf("The stack is empty.");
    else {
        printf("Printing elements in the stack..\n");
        while (S->next != NULL) {
            printf("%d\n", S->data);
            S = S->next;
        }
        printf("%d\n", S->data);
    }
}
```

A program that creates the stack and carries out stack operations.

```
void Error(char *message);
int main()
{
    Item *myStack = NULL;
    Push(4, &myStack);
    Push(5, &myStack);
    Push(7, &myStack);
    print_stack(myStack);
    Pop(&myStack);
    print_stack(myStack);
    printf("Element on top is \n%d", Top(myStack));
    return 0;
}
```

E3)
```
int ishigher(char op1, char op2)
{
    if( (op1 == '+' || op1 == '-') && (op2 == '*'))
        return(0);
    else
        return(1);
}
```

E4)
```
/*Program-13.3. A program to evaluate an
expression in RPN. File name: unit13-evaluate-RPN.c*/
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#define MAX_STRING 20
typedef struct node {
    int data;
    struct node *next;
} Item;
int IsEmpty(Item * S);
void EmptyStack(Item ** S);
void Push(int x, Item ** S);
int Top(Item * S);
void Pop(Item ** S);
void print_stack(Item * S);
void Error(char *message);
int ishigher(char op1, char op2);
```

```
char *infix2rpn(char *input);
int eval(char op, int a, int b);
int main()
{
    int i,j,op1,op2;
    char out[MAX_STRING];
    Item *myStack = NULL;
    infix2rpn(out);
    printf("RPN form is\n");
    for(i=0;out[i] != '\n';i++)
        printf("%c",out[i]);
    printf("\n");
    for(i = 0; out[i] != '\n';i++){
        if(isdigit(out[i]))
            Push(out[i]-'0',&myStack);
        else{
            op1= Top(myStack);
            Pop(&myStack);
            op2= Top(myStack);
            Pop(&myStack);
            j=eval(out[i],op1,op2);
            Push(j,&myStack);
        }
    }
        printf("Value is %d",Top(myStack));
    return 0;
}
int IsEmpty(Item * S)
{
    if (S != NULL)
        return 0;
    else
        return 1;
}
void EmptyStack(Item ** S)
{
    Item *current = *S;
    if (S == NULL)
        printf("Error! No stack to empty!");
    else
        while (!IsEmpty(current))
            Pop(S);
}
void Push(int x, Item ** S)
{
    Item *tmp;
    tmp = malloc(sizeof(Item));
    if (tmp == NULL) {
        Error("Out of space!");
        exit(1);
    } else {
        tmp->data = x;
        tmp->next = *S;
        *S = tmp;
    }
}
int Top(Item * S)
{
    if (!IsEmpty(S))
        return S->data;
    printf("Empty Stack!");
    return 0;
```

```
}
void Pop(Item ** S)
{
    Item *current, *FirstCell;
    current = *S;
    if (IsEmpty(*S)) {
        printf("Empty Stack");
        return;
    } else {
        FirstCell = current;
        *S = current->next;
        free(FirstCell);
    }
}
void print_stack(Item * S)
{
    if (IsEmpty(S))
            printf("The stack is empty.");
    else {
        printf("Printing elements in the stack..\n");
        while (S->next != NULL) {
            printf("%d\n", S->data);
            S = S->next;
        }
        printf("%d\n", S->data);
    }
}
void Error(char *message)
{
    fprintf(stderr,"Error! %s\n",message);
}
int ishigher(char op1, char op2)
{
    if( (op1 == '+' || op1 == '-') && (op2 == '*'))
        return(0);
    else
        return(1);
}
char *infix2rpn(char *out)
{
    Item *S=NULL;
    char c;
    int i=0;
    while((c=getchar()) != '\n'){
        if (c == ' ')
            continue;
        if(isdigit(c))
            out[i++]=c;
        else if(IsEmpty(S))
            Push(c,&S);
        else{
            while(!IsEmpty(S) && ishigher(Top(S),c)){
                out[i++]=Top(S);
                Pop(&S);
            }
            Push(c,&S);
        }
    }
    while(!IsEmpty(S)){
        out[i++]=Top(S);
        Pop(&S);
    }
```

```
    out[i]='\n';
    return(0);
}
int eval(char op, int a, int b)
{
    switch (op){
        case('+'):
            return(a + b);
        case('-'):
            return(b - a);
        case('*'):
            return(a*b);
    }
}
```