Module -2 - Stacks and Queues

2.1 *Stacks:* Definition

A **stack** is a list of elements in which an element may be inserted or deleted only at one end, called the **top** of the stack i.e. it follows the **last-in first-out** (**LIFO**) principle. This means that elements are removed from a stack in the reverse order of that in which they were inserted into the stack.

The two basic operations associated with a stack are:

i) **Push** or **Add** – used to insert an element into a stack
ii) **Pop** or **Remove** – used to delete an element into a stack

Everyday examples of a stack: a pack of cards, a stack of dishes, a stack of pennies, a stack of folded towels (Fig. 2.1).

Fig. 2.1 Examples of stack

The stack is used to solve a few of the general problems like:

1. Tower of Hanoi
2. N-Queens Problem
3. Infix to Prefix Conversion

Since an item may be added or removed only from the top of a stack, the last item to be added to a stack is the first item to be removed. So, stacks are also called **last-in first-out** (**LIFO**) lists. Other names used for stacks are **piles** and **push-down lists**.

Given a stack S = (a_0, a_1, ... , a_{n-1}), a_0 is the bottom element, a_{n-1} is the top element and a_i is on top of element a_{i-1}, 0<i<n.

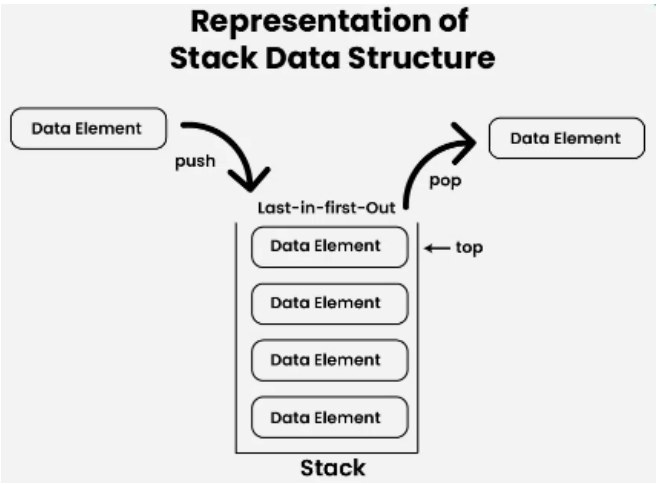

Fig. 2.2 Representation of a stack

Fig. 2.2 shows the representation of a stack.

2.2 Working of Stack

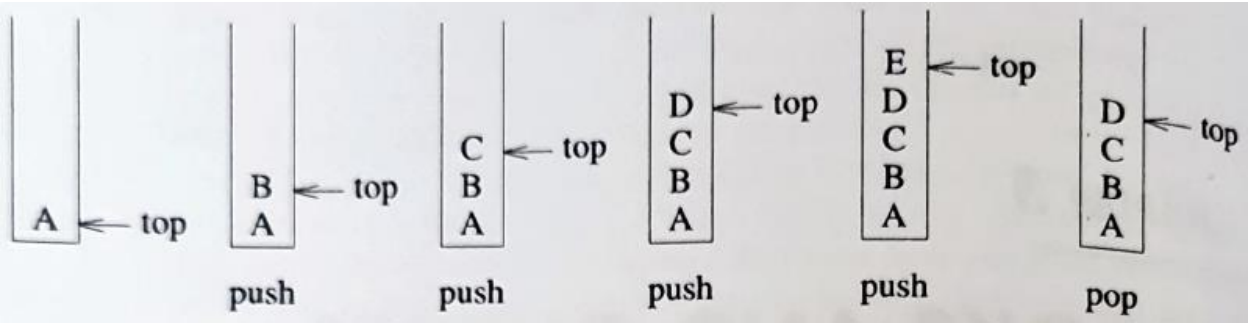

Fig. 2.3 Inserting and deleting elements in a stack

Fig. 2.3 illustrates the sequence of operations on a stack. When an element is "pushed" onto the stack, it becomes the first item that will be "popped" out of the stack. To reach the oldest entered item, you must pop all the previous items.

2.3 Basic Operations on Stacks

The operations associated with a stack are:.

- CREATE(*S*) which creates *S* as an empty stack;
- PUSH(*i,S*) which inserts the element *i* onto the stack *S* and returns the new stack; If the stack is full then the overflow condition occurs.
- POP(*S*) which removes the top element of stack *S* and returns the new stack; If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- TOP(*S*) which returns the top element of stack *S*;
- ISEMPTY(*S*) which returns true if *S* is empty else false;
- ISFULL(*S*) which returns true if *S* is full else false;

2.4 Implementing Stack in C

In C, stacks can be represented using structures, pointers, arrays, or linked lists.

Array Representation of Stacks

2.4.1 Creation of Stack

Stacks may be represented in the computer in various ways, usually by means of

i) an array (one-dimensional array), named STACK;
ii) a pointer variable TOP, which contains the location of the top element of the stack and
iii) a variable MAX_STACK_SIZE which gives the maximum number of elements that can be held by the stack.

The condition TOP = -1 will indicate that the stack is empty.

```
#define MAX_STACK_SIZE 100      /* maximum stack size */
int stack[MAX_STACK_SIZE];      /* stack creation */
```

```
int top = -1,                    /* empty stack */
void push(int item);             /* insert an element */
int pop();          /* delete an element */
void display();     /* displays all the elements of the stack */
```

2.4.2 push(): Stack Insertion

push() is an operation that inserts elements into the stack. Fig. 2.4 illustrates the working of push() operation.

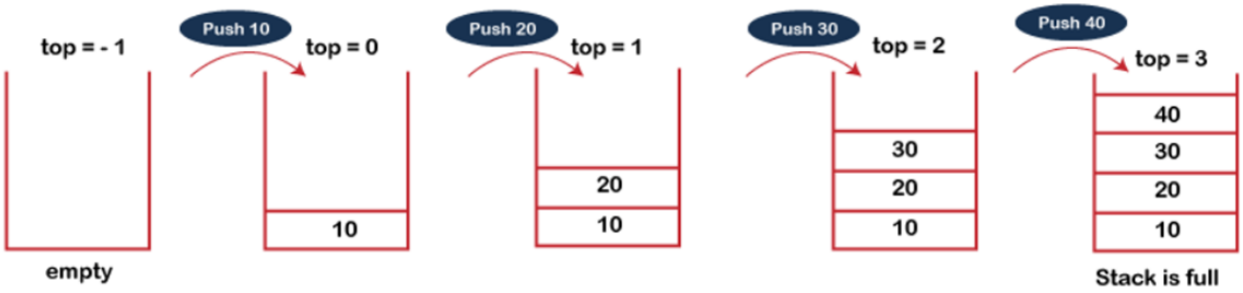

Fig. 2.4 Working of push() operation

Algorithm

1. Checks if the stack is full.
2. If the stack is full, produces an error and exit.
3. If the stack is not full, increments top to point next empty space.
4. Adds data element to the stack location, where top is pointing.
5. Returns success.

Code:

```
void push(int item)
{
/* add an item to the global stack */
   if (top >= MAX_STACK_SIZE – 1)
      printf("Overflow error!!! Item cannot be added to the stack.");
   else
   {
      ++top;
      stack[top] = item;
   }
}
```

2.4.3 Stack Deletion: pop()

pop() is a stack operation which removes elements from the stack.

Fig. 2.5 illustrates the working of pop() operation.

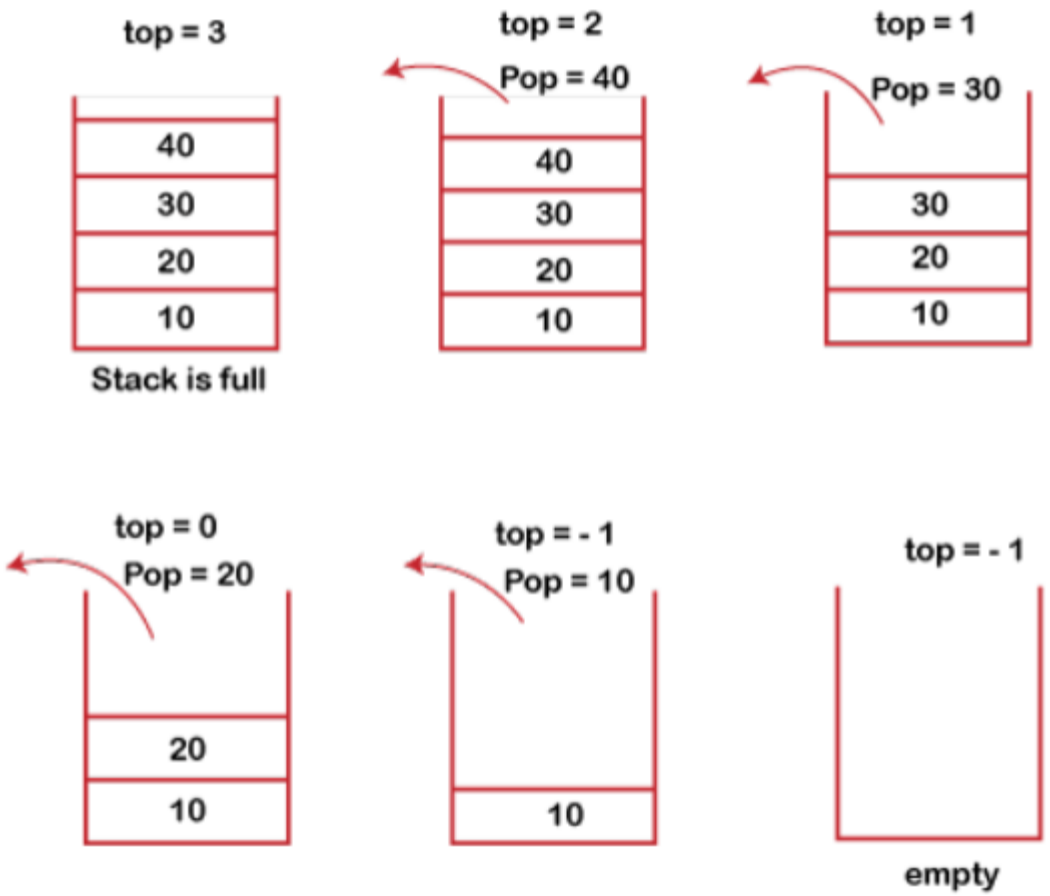

Fig. 2.5 Working of pop() operation

Algorithm

1. Checks if the stack is empty.
2. If the stack is empty, produces an error and exit.
3. If the stack is not empty, accesses the data element at which top is pointing.
4. Decreases the value of top by 1.
5. Returns success.

Code

```
int pop()
{
/* delete and return the top element from the stack */
     if (top == -1)
     {
          printf("Underflow error!!!\n");
          return -999;
     }
     else
          return stack[top--];
}
```

2.4.4 top(): Retrieving the topmost Element of the stack

top() is an operation retrieves the topmost element within the stack, without deleting it. It is also known as the *peek()* operation. This operation is used to check the status of the stack with the help of the top pointer.

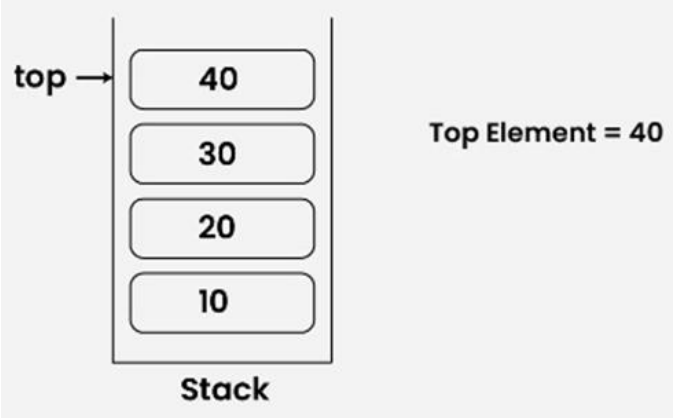

Fig. 2.6 Working of top() operation

Fig. 2.6 illustrates the working of top() operation.

Algorithm

1. Checks if the stack is empty.
2. If the stack is empty, produces an error and exit.
3. If the stack is not empty, returns the data element at which top is pointing.
4. Returns success.

Code

```
int top()
{
/* return the top element of the stack */
     if (top == -1)
     {
          printf("Stack is empty!\n");
          return -999;
     }
     else
          return stack[top];
}
```

2.4.5 isFull(): Verifying whether the Stack is full

isFull() operation checks whether the stack is full. This operation is used to check the status of the stack with the help of top pointer.

Fig. 2.7 illustrates the working of isFull() operation.

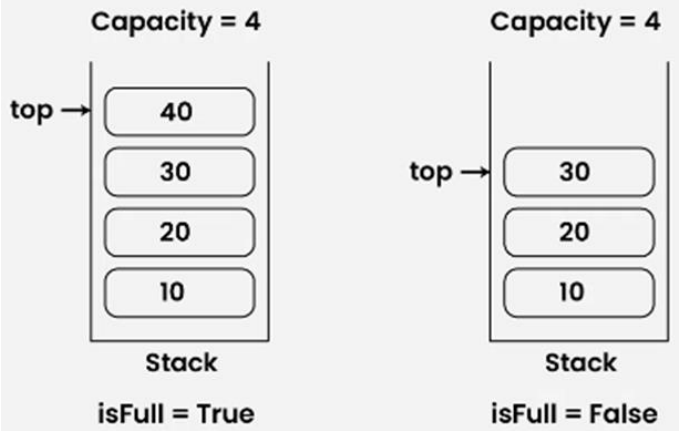

Fig. 2.7 Working of isFull() operation

Algorithm

1. START
2. If the size of the stack is equal to the top position of the stack, the stack is full. Return 1.
3. Otherwise, return 0.
4. END

Code

```
int isfull()
{
  if (top == MAX_STACK_SIZE)
    return 1;
  else
    return 0;
}
```

2.4.6 isEmpty(): Verifying whether the Stack is empty

isEmpty() operation verifies whether the stack is empty. This operation is used to check the status of the stack with the help of top pointer.

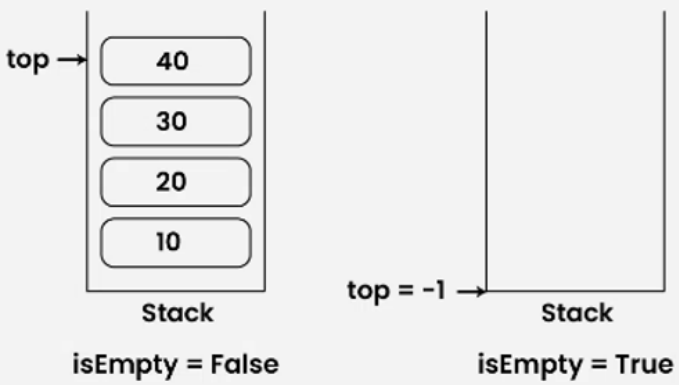

Fig. 2.8 Working of isEmpty() operation

Fig. 2.8 illustrates the working of isEmpty() operation.

Algorithm

1. START
2. If the top value is -1, the stack is empty. Return 1.
3. Otherwise, return 0.
4. END

Code

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

2.4.7 display(): Print all the elements of the stack

```
void display()
{
        int i;
        if(top==-1)
                printf("Stack is empty!");
        else
        {
                printf("Stack elements are:");
                for(i=top;i>=0;i--)
                        printf("%d\t",a[i]);
        }
}
```

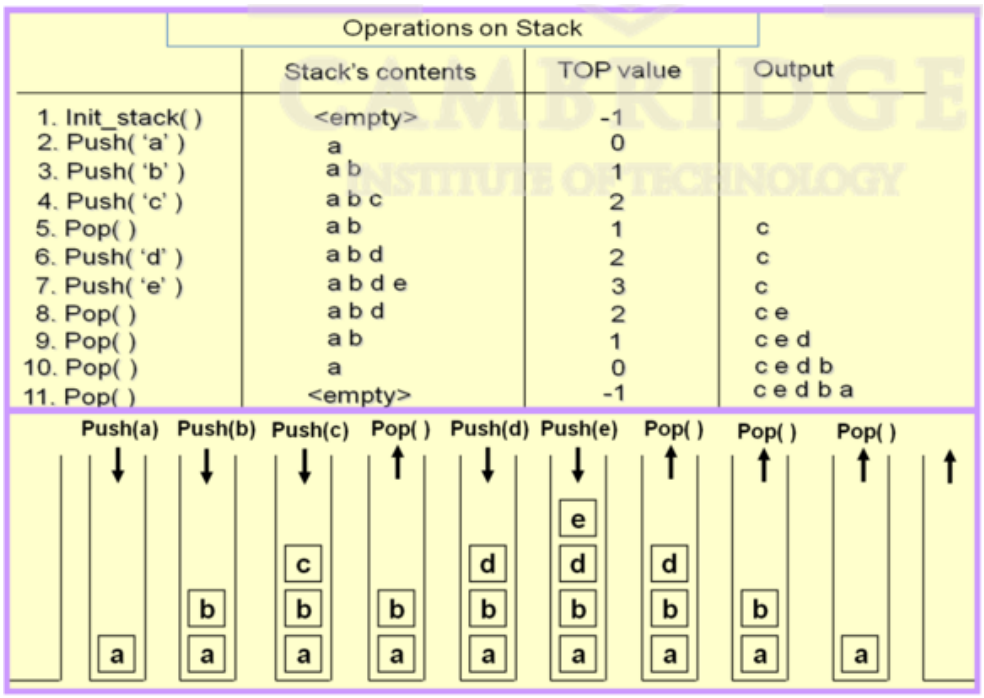

Fig. 2.9 Operation of Stack

Fig. 2.9 shows the operations of a stack.

2.5 ADT STACK

ADT Stack is

objects: a finite ordered list with zero or more elements.

functions:

for all stack ∈ Stack, item ∈ element, maxStackSize ∈ positive integer

Stack CreateS(maxStackSize) ::=
create an empty stack whose maximum size is maxStackSize

Boolean IsFull(stack, maxStackSize) ::=
if (number of elements in stack == maxStackSize)
return TRUE
else return FALSE

Stack Push(stack, item) ::=
if (IsFull(stack)) stackFull
else insert item into top of stack and return

Boolean IsEmpty(stack) ::=
if (stack == CreateS(maxStackSize)
return TRUE
else return FALSE

Element Pop(stack) ::=
if (IsEmpty(stack)) return
else remove and **return** the element at the top of the stack.

Minimizing Overflow

There is an essential difference between underflow and overflow in dealing with stacks. **Underflow** depends exclusively upon the given algorithm and the given input data, and hence there is no direct control by the programmer. **Overflow,** on the other hand, depends upon the arbitrary choice of the programmer for the amount of memory space reserved for each stack, and this choice influences the number of times overflow may occur.

The choice of the amount of memory space reserved for each stack involves a time-space tradeoff. Initially reserving a great deal of space for each stack will decrease the number of times overflow may occur; but this may be an expensive use of the space if most of the space is seldom used. On the other hand, reserving a small amount of space for each stack may increase the number of times overflow occurs; and the time required for resolving an overflow may be more expensive than the space saved.

2.6 Stacks Using Dynamic Arrays

Drawback of array implementation of stack: One should know at compile time how large the stack will become.

- If the set size is small, the stack may run out of space.
- If the set size is large, memory may be wasted.

We can overcome this shortcoming by using a dynamically allocated array for the elements and then increasing the size of this array as needed.

The following are the changes in the array implementation:

```
#define max 10
int *stack;
stack=(int *)malloc(max*sizeof(int));
void push(int ele)
{
        if(top==max-1)
        {
                max=2*max;
                printf("stack overflow so stack is resized and element is pushed\n");
                a=realloc(a,max*sizeof(int));
        }
                a[++top]=ele;
}
```

2.7 Applications of stacks:

1) **System stack**: The system stack is a special stack that is used by a program at run-time to process function calls.
   Whenever a function is invoked, the program creates a structure, referred to as an activation record or a stack frame, and places it on top of the system stack. Initially, the activation record for the invoked function contains only a pointer to the previous stack frame and a return address. The previous stack frame pointer points to the stack frame of the invoking function, while the return address contains the location of the statement to be executed after the function terminates. Since only one function executes at any given time, the function whose stack frame is on top of the system stack is chosen (Fig. 2.10). If this function invokes another function, the local variables, except those declared static, and the parameters of the invoking function are added to its stack frame. A new stack frame is then created for the invoked function and placed on top of the system stack. When this function terminates, its stack frame is removed and the processing of the invoking function, which is again on top of the stack, continues.

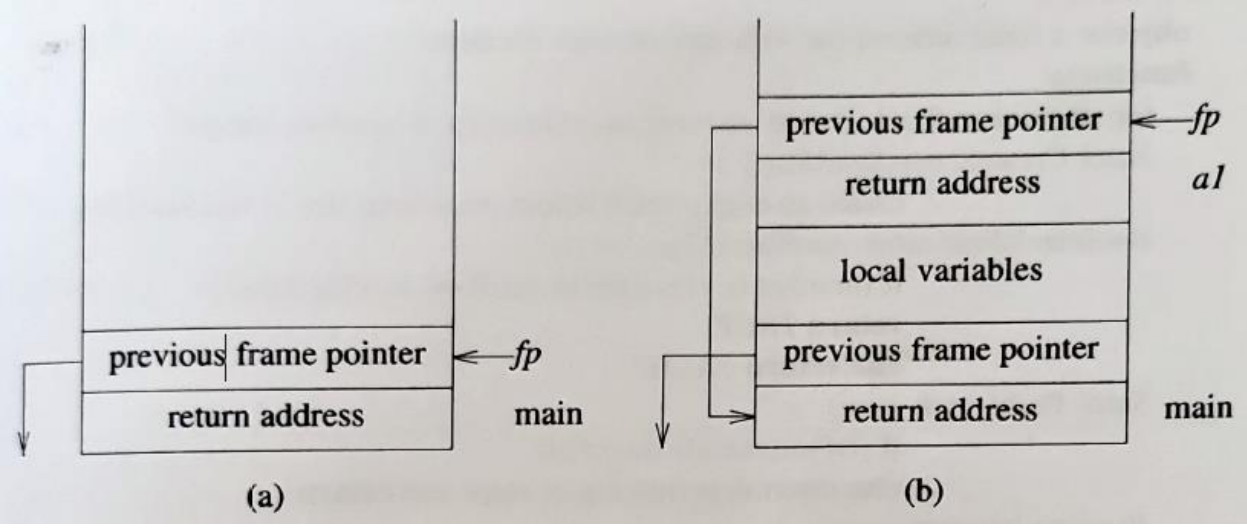

Fig. 2.10 System stack after function call

2) **Implementation of recursive functions**. Recursion function means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.
3) **Balancing of symbols:** Stack is used by compilers for balancing of parenthesis, brackets and braces.
4) **String reversal:** Stack is also used for reversing a string. First, push all the characters of the string in a stack until the null character is reached. After pushing all the characters, take out the characters one by one until we reach the bottom of the stack.
5) **UNDO/REDO:** It can also be used for performing UNDO/REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.

   If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.
6) **DFS (Depth First Search):** This search is implemented on a Graph, and Graph uses the stack data structure.
7) **Backtracking:** Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.
8) **Expression conversion:** Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is:
   ```
   Infix to prefix
   Infix to postfix
   Prefix to infix
   Prefix to postfix
   Postfix to infix
   ```
9) **Evaluation of Postfix expressions.**
10) **Memory management:** The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

2.8 ARITHMETIC EXPRESSIONS: POLISH NOTATION

Binary operators have different levels of precedence. The usual five binary operations have the following levels of precedence:

Highest:	Exponentiation (^)
Next highest:	Multiplication (*) and division (/)
Lowest:	Addition (+) and subtraction (-)

We assume that in any parenthesis-free expression, the operations on the same level are performed from left to right. (However, some languages perform exponentiations from right to left.)

Example: Evaluate the parent

esis-free arithmetic expression:

2 ^ 3 + 5 * 2 ^ 2 – 12 / 6.

Solution: 2 ^ 3 + 5 * 2 ^ 2 – 12 / 6 = 8 + 5*4 – 12 / 6 = 8 + 20 – 2 = 26

Thus, the expression is traversed three times, each time corresponding to a level of precedence of the operations.

For most arithmetic operations the operator symbol is placed between its two operands. This is also called ***infix notation***; Eg. A + B, C-D, E * F, G / H.

We must distinguish between

(A + B) * C and A + (B*C)

by using parenthesis or some operator-precedence. Thus, the order of the operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

Polish Notation:

Named after the polish mathematician Jan Lukasiecthewicz, Polish notation refers to the prefix notation. Here the operator symbol is placed before its two operands. The fundamental property of the Polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression.

Ex.: +AB, -CD, *EF, /GH

Example: Translate the following infix expressions into Polish notation using brackets[] to indicate a partial translation:

i) (A + B) * C = [+AB]*C = *+ABC

ii) A + (B * C) = A + [*BC] = +A*BC

iii) (A+B)/(C-D) = [+AB]/[-CD] = /+AB-CD

iv) (A-B) * (C+D) = [-AB] * [+CD] = *-AB+CD

v) (A+B) / (C+D) – (D*E)
[+AB] / [+CD] – [*DE]
[/+AB+CD] – [*DE]
-/+AB+CD*DE

Reverse Polish notation refers to the notation in which the operator symbol is placed after its two operands. Ex.: AB+, CD- EF*, GH/.

One never needs parenthesis to determine the order of the operations in any arithmetic expression written in reverse Polish notation. This notation is also called as **postfix or suffix notation**.

Example: Convert the following infix expressions into postfix expressions.

a) (A - B) * (C + D) b) (A + B) / (C + D) – (D * E)

Solution:

a) (A - B) * (C + D) [AB-] * [CD+]

AB-CD+*

[AB+] / [CD+] – [DE*]
[AB+CD+/] – [DE*]

b) (A + B) / (C + D) – (D * E)

AB+CD+/DE*-

Infix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.
Example: (A + B) * (C - D)

Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish notation (due to the polish mathematician Jan Lukasiewicz in the year 1920).
Example: * + A B – C D

Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as suffix notation and is also referred to reverse polish notation.
Example: A B + C D - *

The three important features of postfix expression are:

1. The operands maintain the same order as in the equivalent infix expression.
2. The parentheses are not needed to designate the expression unambiguously.
3. While evaluating the postfix expression the priority of the operators is no longer relevant.

The computer usually evaluates an expression written in infix notation in two steps:

i) It converts the expression to postfix expression.
ii) It evaluates the postfix expression.

In both these steps, the stack is the main tool.

2.9 Transforming Infix Expressions into Postfix Expressions

Let Q be an arithmetic expression written in infix notation. We assume that the operators in Q consist of only the following operators:

(i) exponentiation, ^
(ii) multiplication, *
(iii) division, /
(iv) addition, + and
(v) subtraction, -.

Besides these operators and operands, the expression may also contain left and right parenthesis. We also assume that the operators have the usual three levels of precedence, and the operators at the same level of precedence, including exponentiation, are performed from left to right unless otherwise indicated by parenthesis.

Example: Convert the following infix expressions into postfix expressions.

a) (A - B) * (C + D) b) (A + B) / (C + D) – (D * E)

Solution:

a) (A - B) * (C + D)
[AB-] * [CD+]
AB-CD+*

b) (A + B) / (C + D) – (D * E)
[AB+] / [CD+] – [DE*]
[AB+CD+/] – [DE*]
AB+CD+/DE*-

2.9.1 Algorithm

The following algorithm transforms the infix expression Q into its equivalent postfix expression P. The algorithm uses a stack to temporarily hold operators and left parenthesis. The expression P will be constructed from left to right using the operands from Q and the operators removed from the STACK. We begin by pushing a left parenthesis onto STACK and adding a right parenthesis at the end of Q. The algorithm is completed when the STACK is empty.

Algorithm: POLISH(Q, P): Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

STEP 1: Push "(" onto STACK, and add ")" to the end of Q.

STEP 2: Scan Q from left to right and repeat steps 3 to 6 for each element of Q until the STACK is empty.

STEP 3: If an operand is encountered, add it to P.

STEP 4: If a left parenthesis is encountered, push it onto STACK.

STEP 5: If an operator $\otimes$ is encountered, then:

a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than $\otimes$.

b) Add $\otimes$ to STACK.

[End of If structure]

STEP 6: If a right parenthesis is encountered, then:

a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.

b) Remove the left parenthesis. [Do not add the left parenthesis to P.]

[End of If structure]

[End of Step 2 loop.]

STEP 7: EXIT

2.9.2 Examples:

Example: Convert the following arithmetic infix expression Q to postfix expression using the algorithm.

Q: A + (B * C – (D / E ^ F) * G) * H

Solution: First we push "(" onto STACK and then we add ")" to the end of Q and obtain:

Q: A + (B * C – (D / E ^ F) * G) * H)

Symbol Scanned	STACK	Expression P
	(	
A	(	A
+	(+	A
(	(+ (	A
B	(+ (	A B
*	(+ (*	A B
C	(+ (*	A B C
-	(+ (-	A B C *
(	(+ (- (	A B C *
D	(+ (- (	A B C * D

/	(+ (- (/	A B C * D
E	(+ (- (/	A B C * D E
^	(+ (- (/ ^	A B C * D E
F	(+ (- (/ ^	A B C * D E F
)	(+ (-	A B C * D E F ^ /
*	(+ (- *	A B C * D E F ^ /
G	(+ (- *	A B C * D E F ^ / G
)	(+	A B C * D E F ^ / G * -
*	(+ *	A B C * D E F ^ / G * -
H	(+ *	A B C * D E F ^ / G * - H
)		A B C * D E F ^ / G * - H * +

Thus the required postfix equivalent of Q is:

P: A B C * D E F ^ / G * - H * +

Example: Convert the following infix expression into postfix expression using the algorithm.

Q: A – (B / C + (D % E * F) / G)* H)

After adding the),

Q: A – (B / C + (D % E * F) / G)* H))

Applying the algorithm, we get

Infix Character Scanned	Stack	Postfix Expression
	(	
A	(	A
-	(-	A
(	(- (	A
B	(- (	A B
/	(- (/	A B
C	(- (/	A B C
+	(- (+	A B C /
(	(- (+ (	A B C /
D	(- (+ (	A B C / D
%	(- (+ (%	A B C / D
E	(- (+ (%	A B C / D E
*	(- (+ (*	A B C / D E %
F	(- (+ (*	A B C / D E % F
)	(- (+	A B C / D E % F *
/	(- (+ /	A B C / D E % F *
G	(- (+ /	A B C / D E % F * G
)	(-	A B C / D E % F * G / +
*	(- *	A B C / D E % F * G / +
H	(- *	A B C / D E % F * G / + H
)		A B C / D E % F * G / + H* -

Thus, the equivalent postfix expression is:

P: A B C / D E F * % G / + H * -

Example: Convert the following infix expression into postfix expression using the algorithm:

(A+B^D)/(E-F)+G

After adding the sentinel,

(A + B ^ D) / (E – F) + G)

Applying the algorithm, we get

Symbol Scanned	STACK	Expression P
	(	
(	((	
A	((	A
+	((+	A
B	((+	A B
^	((+ ^	A B
D	((+ ^	A B D
)	(	A B D ^ +
/	(/	A B D ^ +
(	(/ (	A B D ^ +
E	(/ (	A B D ^ + E
-	(/ (-	A B D ^ + E
F	(/ (-	A B D ^ + E F
)	(/	A B D ^ + E F -
+	(+	A B D ^ + E F - /
G	(+	A B D ^ + E F - / G
)		A B D ^ + E F - / G +

Thus, the equivalent postfix expression is: A B D ^ + E F - / G +

2.9.3 Program in C

```
#include<stdio.h>
#include<string.h>
char stack[20];
int top=-1;
void push(char s)
{
   stack[++top]=s;
}
char pop()
{
   return stack[top--];
}
```

```
int precd(char s)
{
   switch(s)
   {
        case '^': return 4;
        case '*':
        case '/': return 3;
        case '+':
        case '-': return 2;
        case '(':
        case ')':
        case '#':return  1;
   }
   return 0;
}
void infixToPostfix(char infix[20], char postfix[20])
{
     int i,j=0,n;
     char symbol;
     n = strlen(infix);
     push('#');
     for(i=0;i<n;i++)
     {
        symbol=infix[i];
        switch(symbol)
        {
           case '(': push(symbol);
                     break;
           case ')': while(stack[top]!='(')
                        postfix[j++]=pop();
                        pop();//pop ( bracket
                        break;
           case '^':
           case '*':
           case '/':
           case '+':
           case '-':  while(precd(symbol)<=precd(stack[top]))
                          postfix[j++]=pop();
                      push(symbol);
                      break;
            default:postfix[j++]=symbol;
        }
     }
     while(stack[top]!='#')
        postfix[j++]=pop();
```

```
    postfix[j]='\0';
}
int main()
{
        char infix[20],postfix[20];
        printf("\n Enter infix expression: ");
        gets(infix);
        infixToPostfix(infix,postfix);
        printf("postfix expr is\n %s",postfix);
        return 0;
}
```

2.10 Evaluation of a Postfix Expression

Postfix evaluation algorithm is a simple algorithm that allows us to evaluate postfix expressions. The algorithm uses a stack to keep track of operands and performs arithmetic operations when an operator is encountered.

2.10.1 Algorithm

Algorithm: This algorithm finds the value of an arithmetic expression P written in postfix notation.

Step 1: Add a right parenthesis ")" at the end of P. [This acts as a sentinel.]

Step 2: Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel ")" is encountered.

Step 3: If an operand is encountered, push it on STACK.

Step 4: If an operator $\otimes$ is encountered, then:

a) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.

b) Evaluate B $\otimes$ A.

c) Push the result of (b) back on STACK.

[End of If structure]

[End of Step 2 loop.]

Step 5: Set VALUE equal to the top element on STACK.

Step 6: Exit.

When Step 5 is executed, there should be only one number on STACK.

2.10.2 Examples

Example: Consider the following arithmetic expression P written in postfix notation:

P: 5, 6, 2, +, *, 12, 4, /, -

Commas are used to separate the elements of P so that 5, 6, 2 is not interpreted as the number 562. The equivalent infix expression Q is: 5 * (6 + 2) – 12 / 4.

Adding the sentinel, the expression P becomes: 5, 6, 2, +, *, 12, 4, /, -,)

Evaluating using the algorithm, we get:

Symbol Scanned	STACK
5	5
6	5, 6
2	5, 6, 2
+	5, 8

*	40
12	40, 12
4	40, 12, 4
/	40, 3
-	37
)	

The final value in the STACK, 37, is the value assigned to VALUE when the sentinel ')' is scanned, is the value of P.

Example: Convert the infix expression 9 – ((3 * 4) + 8) / 4 into postfix expression and evaluate the same.

Solution: Infix expression: 9 – ((3 * 4) + 8) / 4
9 – ([3 4 *] + 8) / 4
9 – [3 4 * 8 +] / 4
9 – [3 4 * 8 + 4 /]
9 3 4 * 8 + 4 / -
The postfix expression is: 9 3 4 * 8 + 4 / -
Adding the sentinel, the expression becomes: 9 3 4 * 8 + 4 / - $
Evaluating the expression using the algorithm, we get:

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
-	4

2.10.3 Code in C

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
float stack[20];
int top=-1;
void push(float ele)
{
     stack[++top]=ele;
}
```

```
float pop()
{
     return(stack[top--]);
}
float evaluate(char postfix[50])
{
     int i;
     char sym;
     float op1,op2,result;
     for(i=0;i<strlen(postfix);i++)
     {
          sym=postfix[i];
          if(isdigit(sym))
               push(sym-'0');
          else
          {
               op2=pop();
               op1=pop();
               switch(sym)
               {
                    case '+':push(op1+op2);
                             break;
                    case '-':push(op1-op2);
                             break;
                    case '*':push(op1*op2);
                             break;
                    case '/':push(op1/op2);
                             break;
                    case '%':push((int)op1%(int)op2);
                             break;
                    case '^':push(pow(op1,op2));
                             break;
                    default:printf("invalid postfix expression");
                            exit(0);
               }
          }
     }
     result = pop();
     return(result);
}
int main()
```

```
{
      char postfix[50];
      float ans;
      printf("enter the postfix expression\n");
      gets(postfix);
      ans = evaluate(postfix);
      printf("the result is %f\n",ans);
}
```

2.11 Multiple Stacks

We consider only sequential mappings of stacks into a single array, stack, of size MAX_STACK_SIZE.

2.11.1 Representation of two stacks in a Single Array

If we have only two stacks to represent, the solution is simple.

We use a single array named as 'stack'. Stack1 starts from index 0 while Stack2 starts from index n-1 i.e. both the stacks start from the extreme ends.

Stack1 extends in the right direction i.e. towards MAX_STACK_SIZE - 1, and Stack2 extends in the left direction i.e. towards 0, shown in Fig.2.11. With this representation, we can efficiently use all the available space.

If we push 'a' into stack1 and 'q' into stack2 shown in Fig. 2.11.

Fig. 2.11 Implementation of two stacks in a single array

In this case, the stack overflow condition occurs only when **top1 + 1 = top2**.

This approach provides a space-efficient implementation. This means that when the array is full, then only it will show the overflow error.

2.11.2 Implementation in C

/* C Program to Implement two Stacks using a Single Array and Check for Overflow and Underflow */

```
#include <stdio.h>
#define MAX_STACK_SIZE 20
int stack[MAX_STACK_SIZE];  // declaration of array type variable.
int top1 = -1;
int top2 = MAX_STACK_SIZE;
//Function to push data into stack1
void push1 (int data)
{
// checking the overflow condition
    if (top1 < top2 - 1)
    {
```

```
        top1++;
        stack[top1] = data;
    }
    else
        printf ("Stack is full");
}
// Function to push data into stack2
void push2 (int data)
{
    // checking overflow condition
    if (top1 < top2 - 1)
    {
        top2--;
        stack[top2] = data;
    }
    else
        printf ("Stack is full..\n");
}

//Function to pop data from the Stack1
void pop1 ()
{
    // Checking the underflow condition
    if (top1 >= 0)
    {
        int popped_element = stack[top1];
        top1--;
        printf ("%d is being popped from Stack 1\n", popped_element);
    }
    else
        printf ("Stack is Empty \n");
}
// Function to remove the element from the Stack2
void pop2 ()
{
    // Checking underflow condition
    if (top2 < MAX_STACK_SIZE)
    {
        int popped_element = stack[top2];
        top2++;
        printf ("%d is being popped from Stack 1\n", popped_element);
    }
    else
        printf ("Stack is Empty!\n");
}
```

```
//Function to Print the values of Stack1
void display_stack1 ()
{
    int i;
    for (i = top1; i >= 0; --i)
        printf ("%d ", stack[i]);
    printf ("\n");
}
// Function to print the values of Stack2
void display_stack2 ()
{
    int i;
    for (i = top2; i < MAX_STACK_SIZE; ++i)
        printf ("%d ", stack[i]);
    printf ("\n");
}
int main()
{
    int stack[MAX_STACK_SIZE];
    int i;
    int num_of_ele;
    printf ("We can push a total of 20 values\n");
    //Number of elements pushed in stack 1 is 10
    //Number of elements pushed in stack 2 is 10
    // loop to insert the elements into Stack1
    for (i = 1; i <= 10; ++i)
    {
        push1(i);
        printf ("Value Pushed in Stack 1 is %d\n", i);
    }
    // loop to insert the elements into Stack2.
    for (i = 11; i <= 20; ++i)
    {
        push2(i);
        printf ("Value Pushed in Stack 2 is %d\n", i);
    }
    //Print Both Stacks
    display_stack1 ();
    display_stack2 ();
    //Pushing on Stack Full
    printf ("Pushing Value in Stack 1 is %d\n", 11);
    push1 (11);
    //Popping All Elements from Stack 1
    num_of_ele = top1 + 1;
    while (num_of_ele)
```

```
    {
        pop1 ();
        --num_of_ele;
    }
    // Trying to Pop the element From the Empty Stack
    pop1 ();
    return 0;
}
```

Output:

```
We can push a total of 20 values
Value Pushed in Stack 1 is 1
Value Pushed in Stack 1 is 2
Value Pushed in Stack 1 is 3
Value Pushed in Stack 1 is 4
Value Pushed in Stack 1 is 5
Value Pushed in Stack 1 is 6
Value Pushed in Stack 1 is 7
Value Pushed in Stack 1 is 8
Value Pushed in Stack 1 is 9
Value Pushed in Stack 1 is 10
Value Pushed in Stack 2 is 11
Value Pushed in Stack 2 is 12
Value Pushed in Stack 2 is 13
Value Pushed in Stack 2 is 14
Value Pushed in Stack 2 is 15
Value Pushed in Stack 2 is 16
Value Pushed in Stack 2 is 17
Value Pushed in Stack 2 is 18
Value Pushed in Stack 2 is 19
Value Pushed in Stack 2 is 20
10 9 8 7 6 5 4 3 2 1
20 19 18 17 16 15 14 13 12 11
Pushing Value in Stack 1 is 11
Stack is full10 is being popped from Stack
9 is being popped from Stack 1
8 is being popped from Stack 1
7 is being popped from Stack 1
6 is being popped from Stack 1
5 is being popped from Stack 1
4 is being popped from Stack 1
3 is being popped from Stack 1
2 is being popped from Stack 1
1 is being popped from Stack 1
Stack is Empty
```

2.11.3 Representation of multiple stacks in a Single Array

Representing more than two stacks within the same array poses problems since we no longer have an obvious point for the bottom element of each stack.

Assuming that we have n stacks, we can divide the available memory into n segments. This initial division may be done in proportion to the expected sizes of the various stacks, if this is known. Otherwise, we may divide the memory into equal segments.

Assume that i refers to the stack number of one of the n stacks. To establish this stack, we must create indices for both the bottom and top positions of this stack. The convention we use is that

boundary [i], 0 ≤ i < MAX_STACKS, points to the position immediately to the left of the bottom element of stack i,

top[i], 0 ≤ i < MAX_STACKS, points to the top element of stack i.

Stack i is empty iff boundary[i] = top[i].

The relevant declarations are:

```
#define MAX_STACK_SIZE 100 /* size of memory */
#define MAX_STACKS 10     /* max number of stacks plus 1 */

/* global memory declaration */
element memory [MAX_STACK_SIZE];
int top[MAX_STACKS] ;
int boundary[MAX_STACKS ];
int n; /* number of stacks entered by the user */
```

To divide the array into roughly equal segments we use the following code:

```
top[0] = boundary[0] = -1;
for (j = 1; j < n; j++)
     top[jJ = boundary[j] = (MAX_STACK_SIZE /n) * j - 1;
boundary[n] = MAX_STACK_SIZE - 1;
```

Fig.2.12 shows this initial configuration.

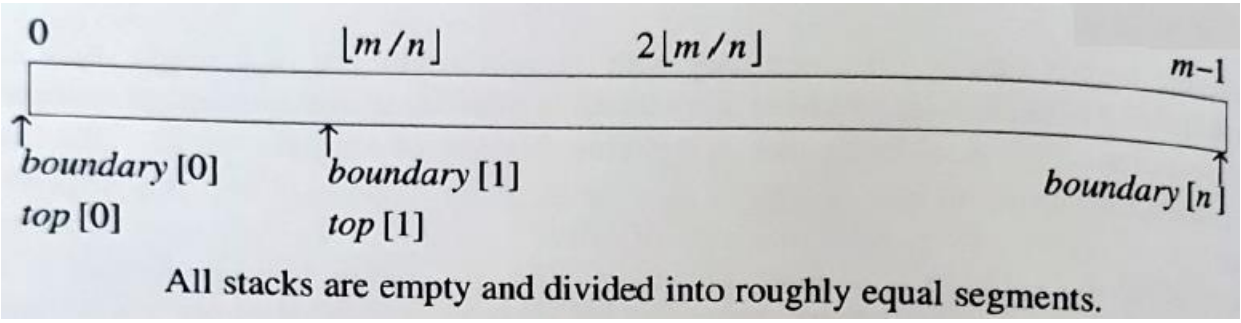

Fig. 2.12 Initial configuration for n stacks in memory[m].

In Fig.2.12, n is the number of stacks entered by the user,

n < MAX_STACKS, and

m = MAX_STACK_SIZE.

Stack i can grow from boundary[i] to boundary [i +1] before it is full.

Since we need a boundary for the last stack, we set boundary[n] to MAX_STACK_SIZE.

Program to add an item to the i^{th} stack

```
void push(int i, element item)
{
    /* add an item to the ith stack */
```

```
    if (top[i] == boundary[i+1] )
        stackFull(i);
    memory[++top[i]] = item;
}
```

Program to delete an item from the i^{th} stack

```
element pop(int i)
{
    /* remove top element from the ith stack */
    if (top[i] == boundary[i])
        return stackEmpty(i) ;
    return memory[top[i] --] ;
}
```

2.12 Recursion

Recursion is an important concept in Computer Science. Many algorithms can be best described in terms of recursion.

2.12.1 Definition

Suppose P is a procedure containing either a call statement to itself or a call statement to a second procedure that may eventually result in a call statement back to the original procedure P, then P is called a ***recursive procedure***. Since a recursive procedure should not run indefinitely, it must have the following two properties:

1) There must be certain criteria, called **base criteria**, for which the procedure does not call itself.
2) Each time the procedure calls itself (directly or indirectly), it must be closer to the base criteria.

A recursive procedure with these two properties is said to be **well-defined**.

The steps of a recursive program are:

Step 1: Specify the base case which will stop the function from making a call to itself.
Step 2: Check to see whether the current value being processed matches with the value of the base case. If yes, process and return the value.
Step 3: Divide the problem into smaller or simpler sub-problems.
Step 4: Call the function from each sub-problem.
Step 5: Combine the results of the sub-problems.
Step 6: Return the result of the entire problem.

Suppose P is a recursive procedure. When an algorithm or program that contains P is run, a **level number** is associated with each execution of procedure P. The original execution of P is assigned level 1; each time P is executed because of a recursive call, its level is one more than the level of execution that has made the recursive call.

The **depth of recursion** of a recursive procedure P with a given set of arguments refers to the maximum level number of P during its execution.

2.12.2 Factorial Function

The product of the positive integers from 1 to n, inclusive, is called 'n factorial'. It is denoted by n!. It is defined for all nonnegative integers.

$n! = 1 . 2 . 3 \ldots (n-2) (n-1) n$

Also $0! = 1$.

$0! = 1$
$1! = 1$

2! = 1 . 2 = 2
3! = 1 . 2. 3 = 6
n! = n . (n-1)!

Accordingly, the factorial function may also be defined as:

a) If n = 0, then n! = 1.
b) If n > 0, then n! = n.(n-1)!

In this definition of n!,
i) It refers to itself.
ii) The value of n! is explicitly given when n = 0; 0 is the base value.
iii) The value of n! for arbitrary n is defined in terms of a smaller value of n which is closer to the base value 0.

So, the procedure is well-defined.

Example:

Calculate 4! Using the recursive definition, this calculation requires nine steps.

Step 1: 4! = 4 . 3!
Step 2: 3! = 3 . 2!
Step 3: 2! = 2 . 1!
Step 4: 1! = 1 . 0!
Step 5: 0! = 1
Step 6: 1! = 1 . 1 = 1
Step 7: 2! = 2. 1 = 2
Step 8: 3! = 3 . 2 = 6
Step 9: 4! = 4 . 6 = 24

That is, in step 5 we evaluate the base value 0!. In steps 6 to 9, we backtrack, using 0! we find 1!, using 1! we find 2!, using 2! we find 3! and finally, using 3!, we find 4!.

2.12.3 Drawbacks of a recursive program

The drawbacks/disadvantages of using a recursive program include the following:

- For some programmers and readers, recursion is a difficult concept.
- Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
- Aborting a recursive program in midstream can be a very slow process.
- Using a recursive function takes more memory and time to execute as compared to its nonrecursive counterpart.
- It is difficult to find bugs, particularly while using global variables.

The advantages of recursion pay off for the extra overhead involved in terms of time and space required.

2.13 TOWER OF HANOI

The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883. He was inspired by a legend that tells of a Hindu temple where the puzzle was presented to young priests. At the beginning of time, the priests were given three poles and a stack of 64 gold disks, each disk a little smaller than the one beneath it. Their assignment was to transfer all 64 disks from one of the three poles to another, with two important constraints. They could only move one disk at a time, and they could never place a larger disk on top of a smaller one. The priests worked very efficiently, day and night, moving one disk every second. When they finished their work, the legend said, the temple would crumble into dust and the world would vanish.

The number of moves required to correctly move a tower of 64 disks is $2^{64}-1=18,446,744,073,709,551,615$. At a rate of one move per second, that is 584,942,417,355 years!

2.13.1 Problem statement:

Suppose there are three pegs labeled A, B and C and on peg A there are placed a finite number n of disks with decreasing size (Fig. 2.13). The object of the game is to move the disks from peg A to peg C using peg B as an auxiliary. The rules of the game are as follows:

1) Only one disk may be moved at a time. Specifically, only the top disk on any peg may be moved to any other peg.
2) At no time can a larger disk be placed on a smaller disk.

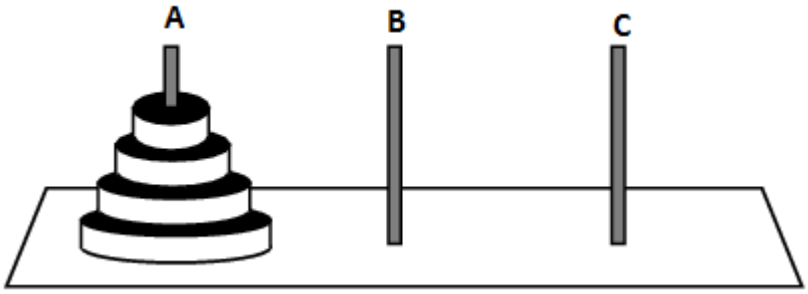

Fig. 2.13 Initial setup of Tower of Hanoi with n = 4

2.12.2 Algorithm

Rather than finding a separate solution, we use the technique of recursion to develop a general solution.

Here is a high-level outline of how to move n disks from the peg A, to peg C, using an auxiliary / intermediate peg, peg B:

1. Move the top n-1 disks from the original peg A to the auxiliary peg B, using the final peg C.
2. Move the remaining disk from the original peg A to the final peg C.
3. Move the n-1 disks from the auxiliary peg B to the final peg C using the original peg A.

Each of the three subproblems listed in steps 1 to 3 may be solved directly or is essentially the same as the original problem using fewer disks.

The only thing missing from the outline above is the identification of a base case. The simplest Tower of Hanoi problem is a tower of one disk. In this case, we need move only a single disk to its final destination i.e. from the original peg A to the final peg C – this is the base case.

The steps outlined above move us toward the base case by reducing the number of disks in steps 1 and 3 (Fig.2.14). Thus, this reduction process yields a recursive solution to the Tower of Hanoi problem.

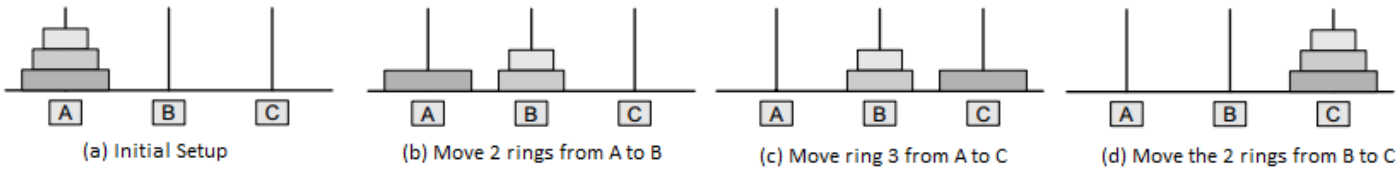

Fig. 2.14 Solution to Tower of Hanoi with n=3

Procedure: TOWER(N, BEG, AUX, END)

This procedure moves the top N disks from the initial peg BEG to the final peg END using the peg AUX as an auxiliary.

STEP 1: If N = 1 then:

a) Print "Move disk" N "from peg " BEG "to peg" END
b) Return

[End of IF structure]

STEP 2: [Move N-1 disks from peg BEG to peg AUX.]
Call TOWER(N-1, BEG, END, AUX).
STEP 3: Print "Move disk" N "from peg " BEG "to peg" END.
STEP 4: [Move N-1 disks from peg AUX to peg END.]
Call TOWER(N-1, AUX, BEG, END).
STEP 5: Return.

2.12.4 Code in C

```
#include<stdio.h>
int count=0;

void TowerofHanoi(int n, char s, char t, char d)
{
	if(n==1)
	{
		printf("move from %c to %c\n",s,d);
		count++;
	}
	else
	{
		TowerofHanoi(n-1, s,d,t);
		printf("move from %c to %c\n",s,d);
		count++;
		TowerofHanoi(n-1, t,s,d);
	}
}

int main()
{
	int n;
	printf("enter the number of disks\n");
	scanf("%d",&n);
	TowerofHanoi(n,'S','T','D');
	printf("no of moves=%d\n",count);
}
```

2.13 Queue - Definition

A queue is a linear list of elements in which insertions can take place only at one end, called the ***rear*** and deletions can take place only at the other one end, called the ***front.***

Queues are also called **first-in-first-out (FIFO)** lists, since the first element in a queue will be the first element out of the queue i.e. the order in which the elements enter a queue is the order in which they leave. (This differs with the stacks, which are last-in-first-out lists.)

Like stacks, queues can be implemented by using either arrays or linked lists.

2.14 Array representation of a queue

The simplest representation of a queue is a one-dimensional array and two variables, front and rear. The front and rear variables point to the position from where deletions and insertions can be done, respectively.

A real-world example of queue can be a single-lane one-way road (Fig. 2.15), where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Fig. 2.15 Single-lane one-way road

Fig. 2.16 shows the representation of a queue.

Fig. 2.16 Representation of a queue

Consider the queue shown in Fig. 2.17. Here, front = 0 and rear = 5.

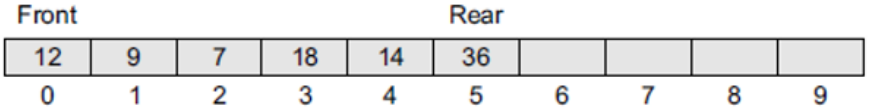

Fig. 2.17 Array representation of a queue

If we want to add one more value to the list, say, if we want to add another element with the value 45, then the rear would be incremented by 1 and the value would be stored at the position pointed by the rear. The queue, after the addition, would be as shown in Fig. 2.18. Here, front = 0 and rear = 6.

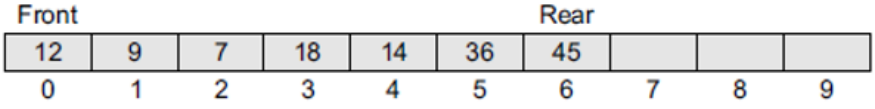

Fig. 2.18 Queue after insertion of a new element

Every time a new element is to be added, we will repeat the same procedure.

However, before inserting an element in the queue, we must check for overflow condition. An ***overflow*** occurs when we try to insert an element into a queue that is already full.

A queue is *full* when rear = MAX – 1, where MAX is the size of the queue, that is MAX specifies the maximum number of elements in the queue. Note that we have written MAX – 1 because the index starts from 0.

Now, if we want to delete an element from the queue, then the value of front will be incremented. Deletions are done only from this end of the queue. The queue after the deletion will be as shown in Fig. 2.19.

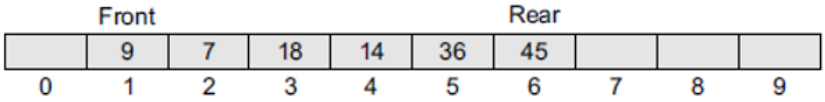

Fig. 2.19 Queue after deletion of an element

Before deleting an element from the queue, we must check for underflow conditions. An ***underflow*** condition occurs when we try to delete an element from a queue that is already empty. If front = NULL and rear = NULL, then there is no element in the queue.

2.15 Abstract data type queue

ADT Queue is

objects: a finite ordered list with zero or more elements.

functions:

for all queue ∈ Queue, item ∈ element, maxQueueSize ∈ positive integer

Queue CreateQ(maxQueueSize) ::=
create an empty queue whose maximum size is maxQueueSize

Boolean lsFullQ(queue, maxQueueSize) ::=
if (number of elements in queue == maxQueueSize)
return TRUE
else return FALSE

Queue AddQ(gueue, item) ::=
if (IsFullQ(queue)) queueFull
else insert item at rear of queue and return queue

Boolean IsEmptyQ(queue) ::=
if (queue == CreateQ(maxQueueSize))
return TRUE
else return FALSE

Element DeleteQ(queue) ::=
if (IsEmptyQ(queue)) **return**
else remove and return the item at front of queue.

ADT 3.2: Abstract data type Queue

2.16 Queue Operations

2.16.1 Basic operations on a queue:

- **Add: Enqueue:** Adds a new element to the queue.
- **Delete: Dequeue:** Removes and returns the first (front) element from the queue.
- **Peek:** Returns the first element in the queue.
- **isEmpty:** Checks if the queue is empty.
- **isFull:** Checks if the queue is full.
- **Size:** Finds the number of elements in the queue.

2.16.2 Declarations

The **declarations** for the one-dimensional array implementation of queue:

```
Queue CreateQ(maxQueueSize) ::=
    #define MAX_QUEUE_SIZE 100 /* maximum queue size */
    typedef struct
    {
        int key;
        /* other fields */
    } element;
    element queue (MAX_QUEUE_SIZE] ;
    int rear =-1;
    int front = -1;
```

2.16.3 Operations

The queue **operations** for the one-dimensional array representation are as follows:

Boolean lsEmptyQ(queue) ::= front == rear

Boolean IsFullQ(queue) ::= rear == MAX_QUEUE_SIZE-1

2.16.4 Queue Insertion Operation: Addq() or Enqueue()

The *addq()* or *enqueue()* is a data manipulation operation that is used to insert elements into the stack. Fig. 2.20 illustrates the addq() or enqueue() operation.

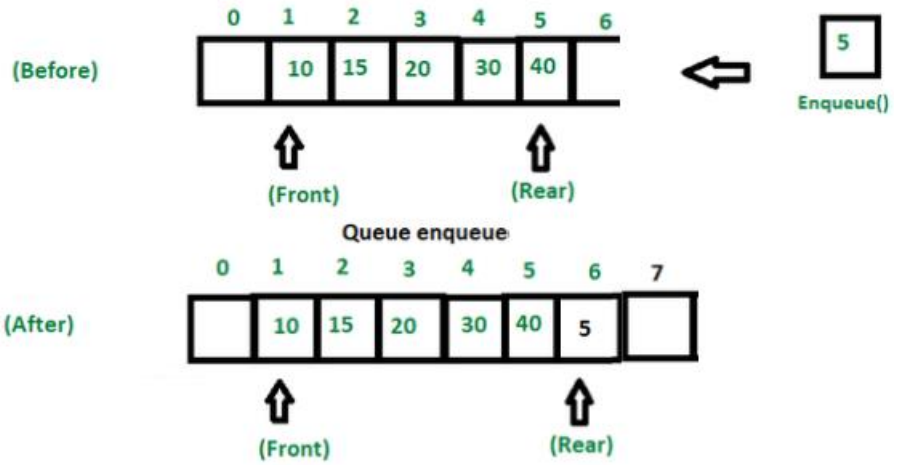

Fig. 2.20 Operation of addq()

Algorithm

1. START
2. Check if the queue is full.
3. If the queue is full, produce overflow error and exit.
4. If the queue is not full, increment rear pointer to point the next empty space.
5. Add data element to the queue location, where the rear is pointing.
6. return success.
7. END

Code

```
void addq(element item)
{ /* add an item to the queue */
    if (rear == MAX_QUEUE_SIZE-1)
        queueFull() ;
```

```
    queue[++rear] = item;
}
```

2.16.5 Queue Deletion Operation: deleteq() or dequeue()

The deleteq() or dequeue() is a data manipulation operation that is used to remove elements from the stack. Fig. 2.21 illustrates the deleteq() or dequeue() operation.

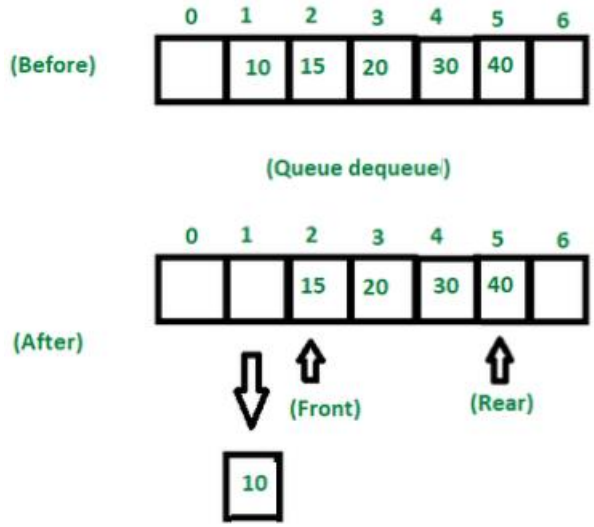

Fig. 2.21 Operation of deleteq() or dequeue()

Algorithm

1. START
2. Check if the queue is empty.
3. If the queue is empty, produce underflow error and exit.
4. If the queue is not empty, access the data where front is pointing.
5. Increment front pointer to point to the next available data element.
6. Return success.
7. END

Code

```
element deleteq()
{ /* remove element at the front of the queue */
    if (front == rear)
        return queueEmpty(); /* return an error key */
    return queue[++front];
}
```

2.16.6 Queue isFull() Operation: isFull()

The isFull() operation verifies whether the queue is full.

Algorithm

1. START
2. If rear equals MAX_QUEUE_SIZE – 1, return true means queue is full.
3. Otherwise return false, means queue is not full.
4. END

Code

```
bool isFull()
{ /* checks if the queue if full */
   return (rear == MAX_QUEUE_SIZE – 1);
```

```
}
```

2.16.7 Queue isEmpty() Operation: isEmpty()

The isEmpty() operation verifies whether the stack is empty.

Algorithm

1. START
2. If front equals rear, return true
3. Otherwise, return false
4. END

Code

```
bool isEmpty()
{    // Checks whether the queue is empty or not:
   return (front == rear-1);
}
```

2.16.8 Queue Peek Operation: peek()

peek() is an operation which is used to retrieve the frontmost element in the queue, without deleting it. This operation is used to check the status of the queue with the help of the pointer.

Algorithm

1. START
2. If the queue is empty produce underflow error and return the most minimum value.
3. Return the front value.
4. END

Code

```
int front(Queue* queue)
{ /* Function to get front of queue */
   if (isEmpty(queue))
      return INT_MIN;
   return queue[front];
}
```