UNIT 3: The Stack

a. Concept and Definition
   - Primitive Operations
   - Stack as an ADT
   - Implementing PUSH and POP operation
   - Testing for overflow and underflow conditions

b. The Infix, Postfix and Prefix
   - Concept and Definition
   - Evaluating the postfix operation
   - Converting from infix to postfix

c. Recursion
   - Concept and Definition
   - Implementation of:
     - ➢ Multiplication of Natural Numbers
     - ➢ Factorial
     - ➢ Fibonacci Sequences
     - ➢ The Tower of Hanoi

Stack

- A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at only one end, called the ***top*** of the stack.
- Unlike array, the definition of the stack provides for the insertion and deletion of items, so that a stack is a dynamic, constantly changing object.
- For insertion, new items are put on the top of the stack in which case the top of the stack moves upward to correspond to the new highest element.
- For deletion, items which are at the top of the stack are removed in which case the top of the stack moves downward to correspond to the new highest element.
- Since items are inserted and deleted in this manner, a stack is also called a *last-in, first-out* (LIFO) list.

Fig: Stack

Primitive Operations on Stack

- When an item is added to a stack, it is ***pushed*** onto the stack, and when an item is removed, it is ***popped*** from the stack.
- Given a stack ***s***, and an item ***i***, performing the operation ***push(s, i)*** adds the item ***i*** to the top of stack ***s***. Similarly, the operation ***pop(s)*** removes the element at the top of ***s*** and returns it as a function value. Thus the assignment operation ***i=pop(s);*** removes the element at the top of ***s*** and assigns its value to ***i***.
- *Note: Because of the push operation which adds elements to a stack, a stack is also called a **pushdown list**.*

Other Operations

- make_empty(s): creates an empty stack ***s***
- is_empty(s): returns true if stack is empty
- is_full(s): returns true if stack is full
- top(s): returns the element which is at top of the stack, but doesn't remove it

- There is no upper bound on the number of items that may be kept in a stack if it is implemented dynamically, i.e. the operation ***push*** is applicable to any stack but if it is implemented statically we can not insert item if it has reached its maximum limit. If we try to insert an item in an already full stack it is called ***stack overflow.***
- If a stack contains a single element and the stack is popped, the resulting stack contains no items, called ***empty stack***, and the operation ***pop*** cannot be applied to ***empty stack***. If we try to pop an item from already empty stack it is called ***stack underflow***

- The operation ***is_empty(s)*** determines whether or not a stack ***s*** is empty. If the stack is empty, ***is_empty(s)*** returns the value ***TRUE***; otherwise it returns the value ***FALSE***.
- The operation ***is_full(s)*** determines whether or not a stack ***s*** is full. If the stack is full, ***is_full(s)*** returns the value ***TRUE***; otherwise it returns the value ***FALSE***.

- Another operation that can be performed on a stack is to determine what the top item on a stack is without removing it.
- The operation ***stacktop(s)*** returns the top element of stack ***s***.
- Similar to operation ***pop***, ***stacktop*** is not defined for an empty stack.
- The result of an illegal attempt to ***pop*** or access an item from an empty stack is called ***underflow***.
- ***Underflow*** can be avoided by ensuring that ***empty(s)*** is **FALSE** before attempting the operation ***pop(s)*** or ***stacktop(s)***.

Stack as an ADT

- Let *eltype* denote the type of the stack element with *STACK(eltype)* being the parameterized stack type with *eltype*.

```
/* value definition */
abstract typedef <<eltype>> STACK(eltype);

        /* operator definition */
        abstract empty(s)
        STACK(eltype) s;
        postcondition empty == (len(s) == 0);

        abstract eltype pop(s)
        STACK(eltype) s;
        precondition empty(s) == FALSE;
        postcondition pop == first(s^1);
                       s == sub(s^1, 1, len(s^1-1));

        abstract push(s, elt)
        STACK(eltype) s;
        eltype elt;
        postcondition s == <elt>+s^1;
```

Representing stacks in C

- An array and a stack are two completely different things (An array is a collection of items and has a finite size whereas the size of a stack is dynamic and changes as elements are pushed or popped).
- However an array can be used as a home for a stack by declaring an array that is large enough for the maximum size of the stack so that during program execution, the stack can grow and shrink within the space reserved for it.
- One end of the array is the fixed bottom of the stack, while the top of the stack constantly shifts as items are pushed and popped.
- Therefore, a field that keeps track of the current position of the top of the stack is needed at each point during program execution.

Representing stacks in C…

- Thus, a stack in C can be declared as a structure containing two members: an ***array*** to hold the elements of the stack, and an ***integer*** to indicate the position of the current stack top within the array.

```
#define MAXSIZE 100
struct stack
    {
            int items[MAXSIZE];
            int top;
    };
```

An actual stack can now be declared as:

```
    struct stack s;
```

Representing stacks in C…

- Note:
  - If the value of *s.top* is 4, there are five elements on the stack: *s.items[0]*, *s.items[1]*, *s.items[2]*, *s.items[3]*, and *s.items[4]*.
  - *When the stack is popped, the value of s.top is changed to 3 to indicate that there are now only 4 elements on the stack and that s.items[3] is the top element.*
  - *When a new item is pushed onto the stack, the value of s.top must be increased by 1 i.e. s.top becomes 5 and the new item is inserted into s.items[5].*

Representing stacks in C…

- To initialize the stack *s* to the empty state, we should initialize **s.top to -1**.
- To determine whether the stack is full or not the condition **s.top==MAXSIZE-1** should be tested.
- Also to determine whether or not a stack is empty, the condition **s.top==-1** should be tested.

Implementing the *pop* operation

- There is possibility of underflow in implementing the *pop* operation, since the user may inadvertently attempt to pop an element from an empty stack.
- To avoid such condition, we should create a function ***pop*** that performs the following three actions:
  1. If the stack is empty, print a warning message and halt execution.
  2. Remove the top element from the stack.
  3. Return this element to the calling program.

```
//C code to implement pop
int pop(struct stack *ps)
{
if(ps->top == -1)
   {
   printf("\n STACK UNDERFLOW");
   exit(1);
   }
return (ps->items[ps->top--]);
/*top item is returned and after that top is
   decremented*/
}
```

Implementing the *push* operation

- A stack is a dynamic data structure that is constantly allowed to grow and shrink and thus change its size whereas an array has a predetermined size.
- The array implementation of stack's *push* operation may thus sometimes outgrow the array size that was set aside to contain the stack.
- This situation occurs when the array is full, i.e. when the stack contains as many elements as the array and an attempt is made to push yet another element onto the stack. The result of such an attempt is called an ***overflow***.

//C code to implement *push*

```
void push(struct stack *ps, int x)
{
if(ps->top == STACKSIZE-1)
    {
    printf("\n STACK OVERFLOW");
    exit(1);
    }
else
    {
    printf("\n Enter value you want to push:");
    scanf("%d", &x);
    ++(ps->top);
    ps->items[ps->top] = x;
    }
}
```

Model Question (2008)

- Define Stack as an ADT. Explain the condition that is to be checked for Push and Pop operations when Stack is implemented using array?

TU Exam Question (2066)

- Write a menu program to demonstrate the simulation of stack operations in array implementation.

```
#define STACKSIZE 100
void push(struct stack *, int);
int pop(struct stack *);
void display(struct stack *);
struct stack
      {
      int items[STACKSIZE];
      int top;
      };
void main()
{
struct stack s;
char ch='y';
char option;
int i;
int x;
s.top=-1;
clrscr();
      while(ch=='y')
      {
      printf("\n What do you want to do?");
      printf("\n1.Push item to the stack");
      printf("\n2.Pop item from the stack");
      printf("\n3.Display stack contents");
      printf("\n4.Exit");
      printf("\n\n Enter your option:\t");
      scanf(" %c", &option);
switch(option)
      {
      case '1':
                  printf(“\n Enter value to push:”)
                  scanf(“%d”, &x);
                   push(&s, x);
                   break;
      case '2':
                  i=pop(&s);
                  printf("\n The popped item is:%d", i);
                                    break;
      case '3':
                  display(&s);
                  break;
      default:
                  exit(1);
      }
printf("\n Do you want to continue(y/n)?:\t");
scanf(" %c", &ch);
      }
getch();
}
```

```
void push(struct stack *ps, int x)
{
if(ps->top == STACKSIZE-1)
    {
    printf("\n STACK
    OVERFLOW");
    exit(1);
    }
else
    ps->items[++(ps->top)] = x;
}


int pop(struct stack *ps)
{
if(ps->top == -1)
    {
    printf("\n STACK
    UNDERFLOW");
    exit(1);
    }
return (ps->items[ps->top--]);
/*top item is returned and after
    that top is decremented*/
}


void display(struct stack *ps)
{
int i;
printf("\n The stack elements
    are:");
for(i=ps->top;i>=0;i--)
    {
    printf("\n|%d|", ps-
    >items[i]);
    }
}
```

INFIX, POSTFIX AND PREFIX

- Consider the sum of A and B. Generally, we apply the ***operator*** "+" to the ***operands*** A and B and write the sum as the expression ***A+B.*** This particular representation is called ***infix***.
- There are two alternate notations for expressing the sum of A and B using the symbols A, B, and +. These are:

  +AB ***(prefix)***

  AB+ ***(postfix)***

INFIX, POSTFIX AND PREFIX…

- The prefixes "pre-", "post-" and "in-" refer to the relative position of the operator with respect to the two operands.
- In prefix notation the operator precedes the two operands, in postfix notation the operator follows the two operands, and in infix notation the operator is between the two operands.

Conversion of Expressions

- The operations involving operator with the highest precedence is converted first and then a portion of that expression is treated as a single operand.
- The precedence of operators is:

◦ Parentheses	()
◦ Exponentiation	$
◦ Multiplication/division	*, /
◦ Addition/subtraction	+, -

↓ **Decreasing order of precedence**

Conversion of Expressions

- When ***unparenthesized operators of the same precedence*** are scanned, the order is assumed to be from ***left to right*** except in the case of ***exponentiation***, where the order is assumed to be from ***right to left***.
- Thus A+B+C means (A+B)+C, whereas A$B$C means A$(B$C).

Convert from infix to prefix and postfix:

a. A+B
b. A+B-C
c. (A+B)*(C-D)
d. A$B*C-D+E/F/(G+H)
e. ((A+B)*C-(D-E))$(F+G)
f. A-B/(C*D$E)

Evaluating a Postfix Expression

- **Algorithm**
  1. The ***postfix*** string is scanned left-to-right one character at a time.
  2. Whenever an operand is read, it is pushed onto the ***opndstack***.
  3. When an operator is read, the top two operands from the stack is popped out into ***op2*** and ***op1***, the ***operator*** is applied in between the two operands (***op1*** ***operator*** ***op2***) and the ***result*** of the operation is pushed back onto the ***opndstack*** (so that it will be available for use as an operand of the next ***operator***).
  4. Pop and display ***opndstack***.

Example: Evaluate 623+-382/+*2$3+

postfix	*op1*	*op2*	*result*	*opndstack*
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
$	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

//Program to evaluate a postfix expression

```
#include<math.h>
#include<string.h>
#define STACKSIZE 100
void push(struct opndstack *,int);
int pop(struct opndstack *);

struct opndstack
   {
   int items[STACKSIZE];
   int top;
   };
```

```
void main()
{
    char postfix[STACKSIZE], ch;
    int i, l;
    int x;
    struct opndstack s;
    int op1,op2;
    int value;
    int result;
    s.top=-1;
    clrscr();
    printf("Enter a valid postfix:");
    gets(postfix);
    l=strlen(postfix);
    for(i=0;i<=l-1;i++)
    {
        if(isdigit(postfix[i]))
        {
            x=postfix[i];
            push(&s,(int)(x-'0'));
        }
else
        {
            ch=postfix[i];
            op2=pop(&s);
            op1=pop(&s);
            switch(ch)
            {
             case '+':

          push(&s,op1+op2);
                                break;
             case'-':

          push(&s,op1-op2);
                                break;
             case'*':

          push(&s,op1*op2);
                                break;
             case'/':

          push(&s,op1/op2);
                                break;
             case'$':

          push(&s,pow(op1,op2));
                                break;
             case'%':

          push(&s,op1%op2);
                                break;
            }
        }
    }
```

```
 result=pop(&s);
printf("\n The final result of postfix
    expression:%d", result);
getch();
}


void push(struct opndstack *ps, int
    x)
{
if(ps->top == STACKSIZE-1)
    {
    printf("\nSTACK OVERFLOW");
    exit(1);
    }
else
    ps->items[++(ps->top)] = x;
}
```

```
int pop(struct opndstack *ps)
{
if(ps->top == -1)
    {
    printf("\n STACK
    UNDERFLOW");
    exit(1);
    }
return (ps->items[ps->top--]);
}
```

Converting an expression from Infix to Postfix

- Algorithm

1. The ***infix*** expression is scanned from left to right one character at a time.
2. If left parentheses i.e. '(' is encountered, push it to ***opstack*** .
3. If operand is encountered, add operand to ***postfix*** string.
4. If operator is encountered, push operator into ***opstack*** **if** the ***opstack*** is empty or **if** the precedence of the current operator on top of ***opstack*** is **smaller** than the currently scanned operator.
5. Whenever right parentheses i.e. ')' is encountered, pop ***opstack*** until a matching left parentheses is found and cancel both.
6. While ***opstack*** is not empty, pop operators from ***opstack*** and add operators to ***postfix*** string.
7. Display ***postfix*** string.

Example: Convert A+B*C to postfix

infix	postfix	opstack
A	A	
+	A	+
B	AB	+
*	AB	+, *
C	ABC	+, *
	ABC*	+
	ABC*+	

Example: Convert (A+B)*C to postfix

infix	postfix	opstack
(		(
A	A	(
+	A	(, +
B	AB	(, +
)	AB+	
*	AB+	*
C	AB+C	*
	AB+C*	

<u>Classwork</u>

Convert

((A-(B+C))*D)$(E+F)

to Postfix.

infix	postfix	opstack
(		(
(		(, (
A	A	(, (
-	A	(, (, -
(	A	(, (, -, (
B	AB	(, (, -, (
+	AB	(, (, -, (, +
C	ABC	(, (, -, (, +
)	ABC+	(, (, -
)	ABC+-	(
*	ABC+-	(, *
D	ABC+-D	(, *
)	ABC+-D*	
$	ABC+-D*	$
(	ABC+-D*	$, (
E	ABC+-D*E	$, (
+	ABC+-D*E	$, (, +
F	ABC+-D*EF	$, (, +
)	ABC+-D*EF+	$
	ABC+-D*EF+$	

TU Exam Question (2065)

- How can you convert from infix to postfix notation.

Why prefix and postfix notations???

- Infix notation is easy to read for *humans*, whereas pre-/postfix notation is easier to parse for a machine.
- The big advantage in pre-/postfix notation is that there never arise any questions like operator precedence.
- The expression is evaluated from left-to-right using postfix and whenever operands are encountered it is pushed onto the stack whereas whenever operator is encountered, two of the operands are popped, the operator is applied in between the two operands and the result is pushed again in the stack.

Why prefix and postfix notations???

- Example:
  - Using infix try to parse A+B*C
    - Push operand A onto stack
    - Save operator + somewhere
    - Push operand B onto stack
    - Now what??? Add A and B ***or*** save another operator *
    - ***Problem:*** Need to know precedence rules and need to look ahead.
  - Using postfix try to parse ABC*+
    - Push operand A onto stack
    - Push operand B onto stack
    - Push operand C onto stack
    - Whenever operator is encountered, pop the two operands from stack, perform the binary operation and push the result onto the stack. Thus at first C and B are popped, then they are multiplied and then the result is pushed onto the stack.
    - Now another operator is encountered, and the above process is repeated again.

Recursion

- **Recursion** in computer science is a problem-solving method where the solution to a problem depends on solutions to smaller instances of the same problem.
- This approach can be applied to many types of problems, and is one of the central ideas of computer science.
- Most computer programming languages support recursion by allowing a function to call itself within the program text.

Recursive function in C

- When a function calls itself directly or indirectly, it is called recursive function.
- Two types:

(i) Direct Recursion (ii) Indirect Recursion

- E.g.

```
void main()
{
printf("This is direct recursion.  Goes infinite\n");
main();
}
```

Recursive function in C…

- E.g.

```
void printline();
void main()
{
printf(" This is not direct recursion.\n");
printline();
}
        void printline()
        {
        printf("Indirect Recursion. Goes Infinite\n");
        main();
        }
```

Problem solving with Recursion

- To solve a problem using recursive method, two conditions must be satisfied:

1) Problem should be written or defined in terms of its previous result.
2) Problem statement must include a terminating condition, otherwise the function will never terminate. This means that there must be an if statement somewhere in the recursive function to force the function to return without the recursive call being executed.

Recursion versus Iteration

Recursion	Iteration
1. A function is called from the definition of the same function to do repeated task.	1. Loops are used to perform repeated task.

2. Recursion is a top-down approach to problem solving: it divides the problem into pieces.
E.g. Computing factorial of a number:
`long int factorial(int n)`
`{`
` if(n==0) return 1;`
`else`
` return (n*factorial(n-1));`
`}`
	2. Iteration is a bottom-up approach: it begins from what is known and from this it constructs the solution step-by-step.
	E.g. Computing factorial of a number:
	`int fact=1;`
	`for(i=1;i<=n;i++)`
	`{`
	`  fact=fact*i;`
	`}`

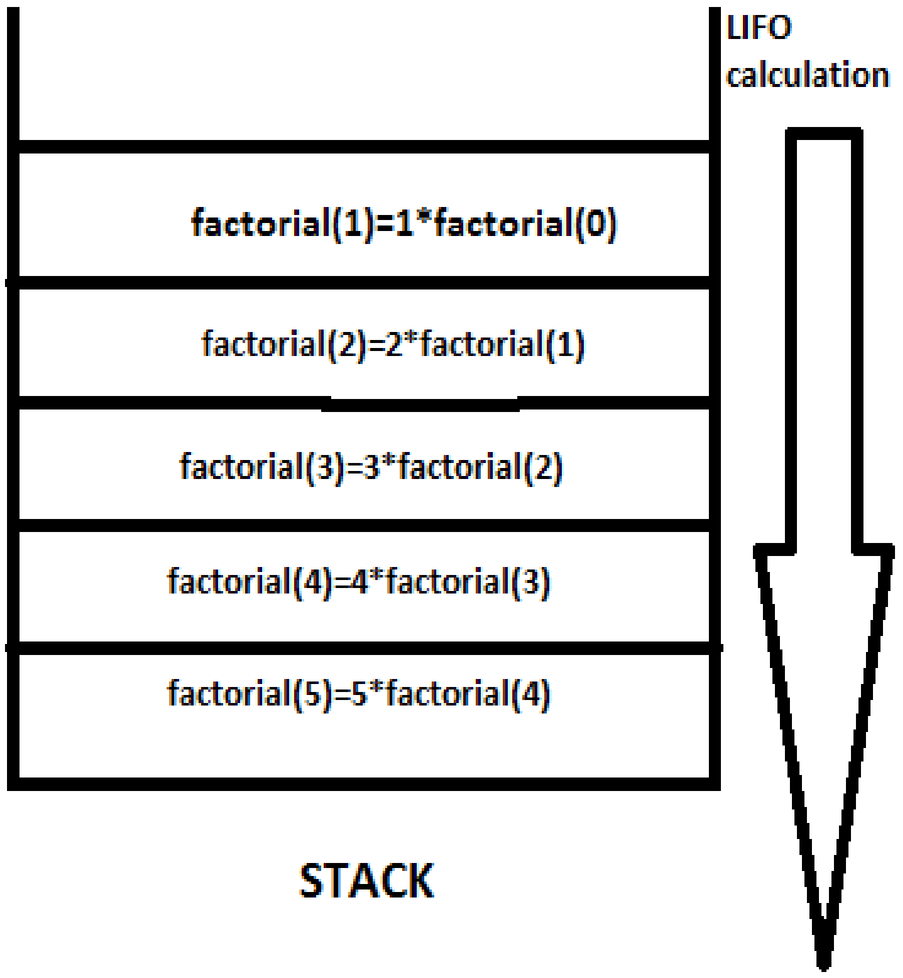
3. Problem to be solved is defined in terms of its previous result to solve a problem using recursion.	3. It is not necessary to define a problem in terms of its previous result to solve using iteration. For e.g. “Display your name 1000 times”
4. In recursion, a function calls to itself until some condition is satisfied.	4. In iteration, a function does not call to itself.
5. All problems cannot be solved using recursion.	5. All problems can be solved using iteration.
6. Recursion utilizes stack.	6. Iteration does not utilize stack.

Use of Stack in Recursion

- Recursion uses stack to keep the successive generations of local variables and parameters of the function in its corresponding calls.
- This stack is maintained by the C system and is invisible to the user (programmer).
- Each time a recursive function is entered, a new allocation of its variables is pushed on top of the stack.
- Any reference to a local variable or parameter is through the current top of the stack.
- When the function returns, the stack is popped, the top allocation is freed, and the previous allocation becomes the current stack top to be used for referencing local variables.
- Each time a recursive function returns, it returns to the point immediately following the point from which it was called.

Implementation of Factorial

```
long int factorial(int n)
{
if(n==0)
    return 1;
else
    return (n*factorial(n-1));
}
    void main()
    {
    int number;
    long int x;
    clrscr();
    printf("Enter a number whose factorial is needed:\t");
    scanf("%d",&number);
    x=factorial(number);
    printf("\n The factorial is:%ld",x);
    getch();
    }
```


LIFO calculation
factorial(1)=1*factorial(0)
factorial(2)=2*factorial(1)
factorial(3)=3*factorial(2)
factorial(4)=4*factorial(3)
factorial(5)=5*factorial(4)
STACK

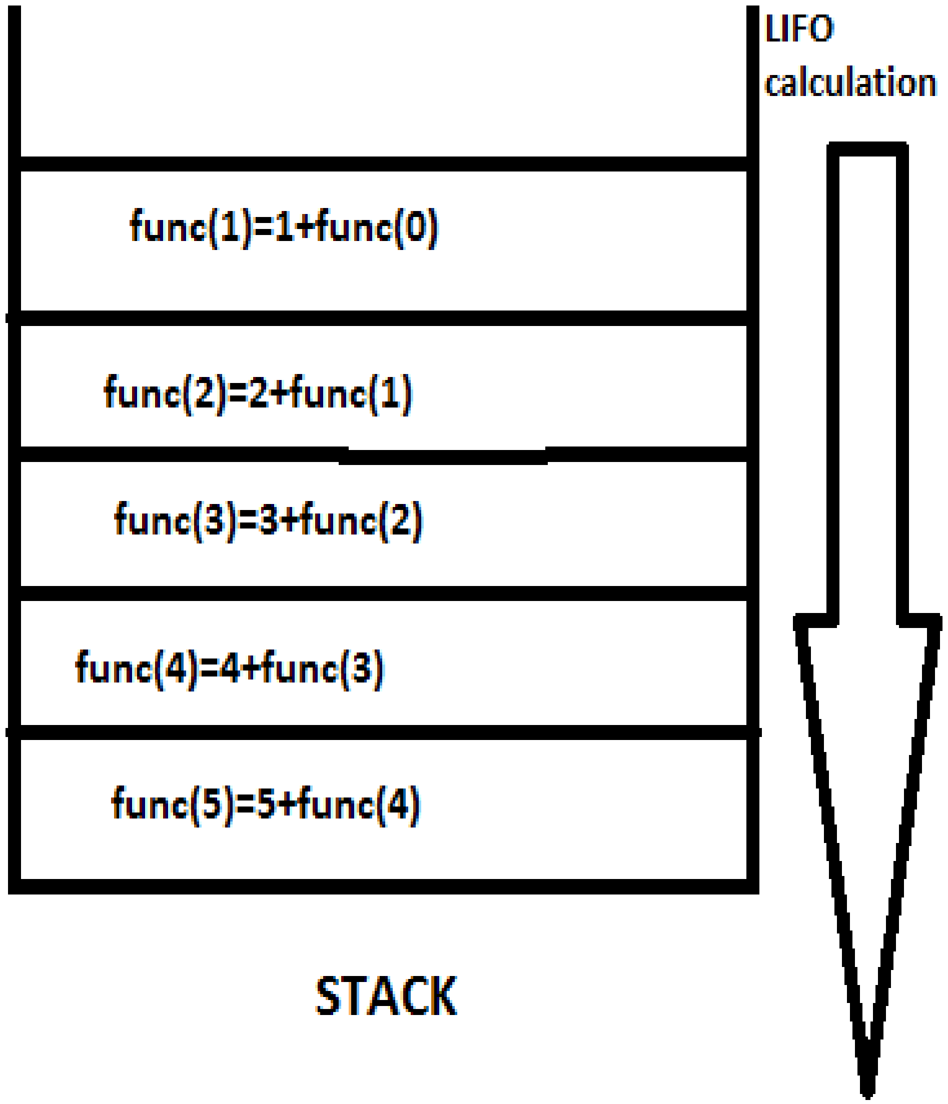
While calculating the factorial of n=5, the else part gets executed and the value 5*factorial(4) is pushed onto the stack. Then the factorial function gets called again recursively with n=4 and 4*factorial(3) is executed and is pushed onto the stack. This process goes on like this. When factorial(1) becomes 1*factorial(0) the factorial function is called again with n=0. When n becomes 0, the function returns 1 and after this the recursive function starts to return in a last-in, first-out manner. The recursive function returns successive values to the point from which it was called in the stack so that the final value returned is 5*24=120.

Model Question (2008)

- Determine what the following recursive C function computes. Write an iterative function to accomplish the same purpose.

```
int func(int n)
{
if(n==0)
        return (0);
return (n + func(n-1));
} /* end func */
```

- The recursive function computes the sum of integers from **0** to **n** where **n** is the input to the function.
- For calculating the sum of integers from 0 to n (with say n=5), the recursive function computes as:
    - **With *n=5*, the *else* part gets executed and the value *5+func(4)* is pushed onto the recursive stack. Then the *func* function is called again recursively with *n=4* and *4+func(3)* is executed and is pushed onto the stack. This process goes on like this. When *func(1)* becomes *1+func(0)*, the *func* function is called again with *n=0*. When n becomes 0, the *func* function returns 0 and after this the recursive function starts to return in a last-in, first-out manner. The recursive function returns successive values to the point from which it was called in the stack so that the final value returned is 5+10=15.**



Iterative Function

int func(int n)
{
int sum=0;
int i;
for(i=0;i<=n;i++)
        sum = sum+i;
return sum;
}

TU Exam Question (2065)

- What do you mean by recursion? Explain the implementation of factorial and fibonacci sequences with example.

Implementation of Fibonacci sequence

//The fibonacci sequence is: 1,1,2,3,5,8,13,…

//The following program computes the nth Fibonacci number

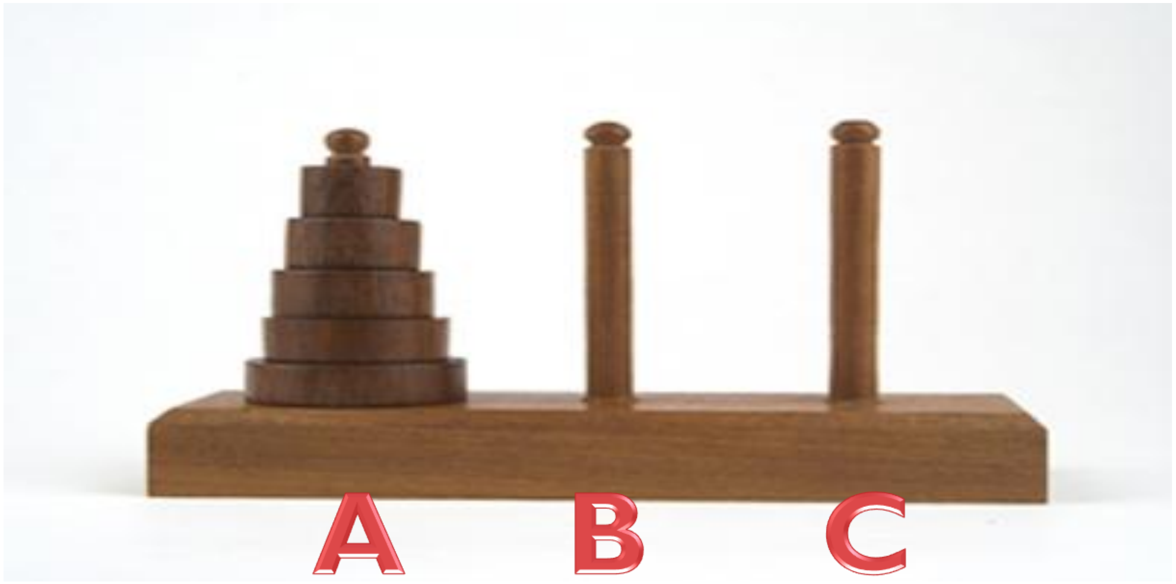
```
int fibo(int n)
{
if(n<=1)
        return n;
else
        return (fibo(n-1)+fibo(n-2));
}

void main()
{
int pos;
int x;
printf("Enter the position of the nth Fibonacci number:");
scanf("%d", &pos);
x=fibo(pos);
printf("\n The %dth Fibonacci number is:%d", pos, x);
}
```

Implementation of Multiplication of Natural Numbers

```
int mult(int a, int b)
{
if(b==0)
     return 0;
else
     return (a+mult(a,--b));
}
void main()
{
int m, n;
int x;
clrscr();
printf("Enter two numbers you want to multiply:");
scanf("%d %d", &m, &n);
x=mult(m, n);
printf("%d*%d=%d", m, n, x);
getch();
}
```

The "Towers of Hanoi" Problem

- There are 3 pegs A, B and C.
- Five disks (***say***) of different diameters are placed on peg A so that a larger disk is always below a smaller disk.
- The aim is to move the five disks to peg C, using peg B as auxiliary.
- Only the top disk on any peg may be moved to any other peg, and a larger disk may never rest on a smaller one.



Origin of “Towers of Hanoi” or “Tower of Brahma” or “Lucas’ Tower” Puzzle

- The puzzle was first publicized by the French mathematician Édouard Lucas in 1883. There is a legend about an Indian temple which contains a large room with three time-worn posts in it surrounded by 64 golden disks. Brahmin priests, acting out the command of an ancient prophecy, have been moving these disks, in accordance with the rules of the puzzle, since that time. The puzzle is therefore also known as the “Tower of Brahma” puzzle. According to the legend, when the last move of the puzzle is completed, the world will end. The temple is said to be in Hanoi, Vietnam.
- If the legend were true, and if the priests were able to move disks at a rate of one per second, using the smallest number of moves, it would take them $2^{64}-1$ seconds or roughly 585 billion years or 18,446,744,073,709,551,615 turns to finish.
- **Solution:** The puzzle can be played with any number of disks, although many toy versions have around seven to nine of them. The puzzle seems impossible to many novices, yet is solvable with a simple algorithm. The number of moves required to solve a Tower of Hanoi puzzle is $\mathbf{2^{n} -1}$, where $\boldsymbol{n}$ is the number of disks.

Basic Idea

- Let us consider the general case of ***n*** disks.
- If we can develop a solution to move ***n-1*** disks, then we can formulate a recursive solution to move all ***n*** disks.
  - In the specific case of 5 disks, suppose that we can move 4 disks from peg A to peg C, using peg B as auxiliary.
  - This implies that we can easily move the 4 disks to peg B also (by using peg C as auxiliary).
  - Now we can easily move the largest disk from peg A to peg C, and finally again apply the solution for 4 disks to move the 4 disks from peg B to peg C, using the now empty peg A as an auxiliary.

Algorithm: Recursive Solution

- To move ***n*** disks from peg A to peg C, using peg B as auxiliary:
  1. If ***n==1***, move the single disk from A to C and stop.
  2. Move the top ***n-1*** disks from A to B, using C as auxiliary.
  3. Move the remaining disk from A to C.
  4. Move the ***n-1*** disks from B to C, using A as auxiliary.

Proof of Correctness:

- If ***n=1***, step1 results the correct solution.
- If ***n=2***, we know we already have a solution for ***n-1=1***, so that topmost disk is put at peg B. Now after performing steps 3 and 4, the solution is completed.
- If ***n=3***, we know we already have a solution for ***n-1=2***, so that 2 disks are at peg B. Now after performing steps 3 and 4, the solution is completed.
- Continuing in this manner, we can show that the solution works for ***n=1,2,3,4,5,…*** up to any value for which we desire a solution.

Implementation of "Towers of Hanoi"

```
void transfer(int, char, char, char);
void main()
{
int n;
clrscr();
printf("\n Input number of disks in peg A:");
scanf("%d", &n);
transfer(n, 'A', 'C', 'B');
getch();
}
    void transfer(int n, char from, char to, char aux)
    {
    if(n==1)
            {
            printf("\n Move disk %d from peg %c to peg %c", n, from, to);
            return;
            }
    transfer(n-1,from,aux,to);
    printf("\n Move disk %d from peg %c to peg %c", n, from, to);
    transfer(n-1,aux,to,from);
    }
```

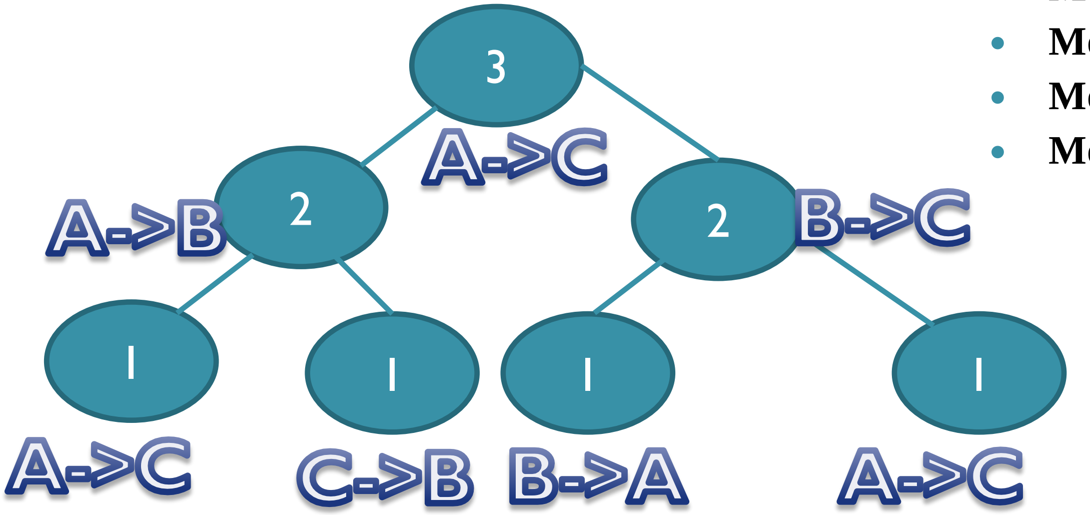
from: the peg from which we are removing disks

to: the peg to which we will take the disks

aux: the auxiliary peg

Tracing with n=3

- **Move disk 1 from peg A to peg C**
- **Move disk 2 from peg A to peg B**
- **Move disk 1 from peg C to peg B**
- **Move disk 3 from peg A to peg C**
- **Move disk 1 from peg B to peg A**
- **Move disk 2 from peg B to peg C**
- **Move disk 1 from peg A to peg C**



•Draw the largest disk node with the largest disk number (disk number starts from 1, with 1 being the smallest disk number) and put it directly to the destination (**A->C**).
•Draw its two children with the second largest node number i.e. 2.
•Now **A->C** can be accomplished only through **A->B** and **B->C**, so put **A->B** as left child and **B->C** as right child.
•Again draw two children of second largest node i.e. 2 and think how we can generate **A->B** and **B->C**……………………………..**Completed**
•Finally perform inorder traversal (left-root-right) of the tree to obtain the appropriate sequence of steps to solve "Tower of Hanoi".
•Note: On left side, we always have A and on right side, we always have C.

TU Exam Question (2065)
Model Question (2008)

- Write and explain the algorithm for Tower of Hanoi.

TU Exam Question (2066)

- Consider the function:

```
void transfer(int n, char from, char to, char temp)
{
if(n>0)
transfer(n-1, from, temp, to);
printf("\n Move Disk %d from %c to %c", n, from, to);
transfer(n-1, temp, to, from);
}
```

Trace the output with the function call:

transfer(3, 'R', 'L', 'C');

Leftovers

Converting an expression from Infix to Prefix

- Assumptions:
  1. We scan one character at a time from right to left.
  2. Correct input is assumed.
  3. Only five binary operators (+, -, *, /, %) are used.
  4. Operators precedence hierarchy is: $ > *, / > +, - > (,)
  5. Only single letter variable names are used.

- Algorithm
  1. The ***infix*** expression is scanned from right to left one character at a time.
  2. While there is data
     i. If right parentheses i.e. ')' is encountered, push it to ***opstack*** .
     ii. If operand is encountered, add operand to ***prefix*** string.
     iii. If operator is encountered, then **if** the ***opstack*** is empty, push operator into ***opstack*** **else if** the precedence of the current operator on top of ***opstack*** is **greater** than the currently scanned operator, then pop and append to ***prefix*** string **else** push into ***opstack*** .
     iv. Whenever left parentheses i.e. '(' is encountered, pop ***opstack*** and append to ***prefix*** string until a matching right parentheses is found, and cancel both parentheses.
  3. While ***opstack*** is not empty, pop operators from ***opstack*** and add operators to ***prefix*** string.
  4. Display reverse of ***prefix*** string .

Trace: ((A-(B+C))*D)$(E+F)

infix	prefix	opstack
)		)
F	F	)
+	F	), +
E	FE	), +
(	FE+	
$	FE+	$
)	FE+	$,)
D	FE+D	$,)
*	FE+D	$,), *
)	FE+D	$,), *,)
)	FE+D	$,), *,),)
C	FE+DC	$,), *,),)
+	FE+DC	$,), *,),), +
B	FE+DCB	$,), *,),), +
(	FE+DCB+	$,), *,)
-	FE+DCB+	$,), *,), -
A	FE+DCB+A	$,), *,), -
(	FE+DCB+A-	$,), *
(	FE+DCB+A-*	$
	FE+DCB+A-*$	

The required prefix string is:
$*-A+BCD+EF

Convert the following infix expression to prefix:
A+(B*C-(D/E$F)*G)*H

Ans: +A*-*BC*/D$EFGH

Evaluating a prefix expression

- **Algorithm**
  1. The ***prefix*** string is scanned right-to-left one character at a time.
  2. Whenever an operand is read, it is pushed onto the ***opndstack***.
  3. When an operator is read, the top two operands from the stack is popped out into ***op1*** and ***op2***, the ***operator*** is applied in between the two operands (***op1*** ***operator*** ***op2***) and the ***result*** of the operation is pushed back onto the ***opndstack*** (so that it will be available for use as an operand of the next ***operator***).
  4. Pop and display ***opndstack***.

Evaluate the prefix expression: $*-A+BCD+EF with A=6, B=1, C=2, D=3, E=2 AND F=1

prefix	op1	op2	result	opndstack
F (1)				1
E (2)				1, 2
+	2	1	3	3
D (3)	2	1	3	3, 3
C (2)	2	1	3	3, 3, 2
B (1)	2	1	3	3, 3, 2, 1
+	1	2	3	3, 3, 3
A (6)	1	2	3	3, 3, 3, 6
-	6	3	3	3, 3, 3
*	3	3	9	3, 9
$	9	3	729	729

The evaluated result of the prefix expression is 729.

Conversion from prefix to postfix

Recursive Algorithm

1. If the prefix string is a single variable, it is its own postfix equivalent
2. Let op be the first operator of the prefix string
3. Find the first operand, opnd1 of the string.Convert it to postfix and call it post1.
4. Find the second operand, opnd2, of the string.Convert it to postfix and call it post2.
5. Concatenate post1, post2, and op.

Conversion from prefix to postfix

```
int prefixToPostfix(char prefix[], char postfix[]) {
    Int nextPrefix1, nextPrefix2;
    char firstChar = prefix[0];
            if (isOperand(firstChar))
            { // must be: <operand>
                        postfix += firstChar;
                         return 1;
            } // must be: <operator><prefix><prefix>
    nextPrefix1 = prefixToPostfix(prefix.substr(1), postfix);
    nextPrefix2 = prefixToPostfix(prefix.substr(nextPrefix1+1),
    postfix);
    postfix += firstChar; // the operator
    return nextPrefix1 + nextPrefix2 + 1;
}
```