FOLLOW US

Website: https://bcanepaltu.com
Page: https://www.facebook.com/bcanepaltu/

QUESTION PAPERS AND NOTES

Unit 1. Concept and Definition of Data Structures

a. Information and its meaning
b. Array in C
c. The array as an ADT
d. One dimensional array
e. Two dimensional array
f. Multi-dimensional array
g. Structure
h. Union
i. Pointer

Prepared By: Ramesh Saud

Data Structure

- In computer science, a **data structure** is a particular way of storing and organizing data in a computer so that it can be used efficiently.
- Various kinds of data structures are ARRAYS, STACKS, QUEUES, LINKED LISTS, TREES, etc. and these different kinds of data structures are suited to different kinds of applications. For example: The data structure ***QUEUE*** is used by printer for printing its assigned jobs.

Information and its meaning

- The definition of information cannot be given precisely but it can be measured.
- The basic unit of information is a **bit** (stands for "binary digit") and the quantity of information can be interpreted in terms of bits.
- Integers, real numbers, characters, etc can be interpreted in terms of bits. Now we need to interpret the bit patterns so that it gives a meaning. For e.g., the bit string 00100110 can be interpreted as the number 38 in binary number system or the character '&'.
- Bit patterns are interpreted using a ***data type***.

Information and its meaning…

- Every computer has a set of "native" ***data types***. The data types offer a method of interpreting information as memory contents in a computer. Further they also specify the range of values legal for that data type and also a set of operations on data of that type.
- The hardware is wired in a way to support those data types too.
- However, when we divorce the "***data type***" from the hardware capabilities of a computer, we can consider a limitless number of data types. The data type in such case becomes abstract (unimplemented) called ***abstract data type (ADT)***.

TU Exam Question (2066)

- Write a short note on ADT.

Information and its meaning…

ADT: An **ADT** is a specification of a set of data and the set of operations that operate on the data. Such data type is ***abstract*** in the sense that it is independent of a concrete implementation.

An ADT has two parts: **a value definition** and **an operator definition**. The value definition defines the collection of values for the ADT and consists of two parts: **a definition clause** and **a condition clause**. Immediately following the value definition comes the **operator definition**. Each operator is defined as an *abstract function* with three parts: a header, the optional preconditions, and the postconditions.

For example: Let us consider the ADT ***RATIONAL***, which corresponds to the mathematical specification of a rational number. A rational number is a number that can be expressed as the quotient of two integers. The operations on rational numbers that we define are the creation of a rational number from two integers, addition, multiplication, and testing for equality.

Information and its meaning…

Rational number as an ADT

The ADT **RATIONAL** is defined as a collection of values that consists of two integers (numerator and denominator), the second of which does not equal to zero.

Operations

- ***makerational(a,b)*****:** create rational number (a/b) with b‡0.
- ***add(x,y)*****:** add two rational numbers with $x=a_0/a_1$ and $y=b_0/b_1$.
- ***mult(x,y)*****:** multiplies two rational numbers $x=a_0/a_1$ and $y=b_0/b_1$.
- ***equal(x,y)*****:** tests for equality of two rational numbers by checking if $a_0*b_1 == b_0*a_1$.

Simulation --- --- --- --- --- --- RATIONAL

```
void makerational(struct RATIONAL *, struct
    RATIONAL *);
void add(struct RATIONAL *, struct RATIONAL
    *);
void mult(struct RATIONAL *, struct RATIONAL
    *);
void equal(struct RATIONAL *, struct
    RATIONAL *);


struct RATIONAL
    {
    int numerator;
    int denominator;
    };
void main()
{
struct RATIONAL a,b;
clrscr();
```

```
scanf("%d/%d", &a.numerator,
    &a.denominator);


    printf("\n Enter numerator and denominator
    of second rational no:");
    scanf("%d/
    %d",&b.numerator,&b.denominator);


    makerational(&a,&b);


    add(&a,&b);


    mult(&a,&b);


    equal(&a,&b);


    getch();
```

```
void makerational(struct RATIONAL *p, struct
    RATIONAL *q)
{
    if(p->denominator==0)
    {
    printf("\n The denominator of first rational
    number cannot be zero.");
    printf("\n Enter other rational number:");
    scanf("%d/%d",&p->numerator,&p-
    >denominator);
    }
    if(q->denominator==0)
    {
    printf("\n The denominator of second
    rational number cannot be zero.");
    printf("\n Enter other rational number:");
    scanf("%d/%d",&q->numerator,&q-
    >denominator);
    }
printf("\n The first rational number is:%d/%d",
    p->numerator, p->denominator);
printf("\n The second rational number
    is:%d/%d", q->numerator, q->denominator);
printf("\n\n\t**********makerational
    complete**********");
}
```

```
void add(struct RATIONAL *p, struct
RATIONAL *q)
{
struct RATIONAL r;
r.denominator = (p->denominator)*(q-
>denominator);
r.numerator = (p->numerator)*(q-
>denominator)+(p->denominator)*(q-
>numerator);
printf("\n The addition of the two rational
numbers is: %d/%d", r.numerator,
r.denominator);
printf("\n\n\t**********addition
complete**********");
}
```

```
void mult(struct RATIONAL *p, struct
RATIONAL *q)
{
struct RATIONAL r;
r.denominator = (p->denominator)*(q-
>denominator);
r.numerator = (p->numerator)*(q-
>numerator);
printf("\n The multiplication of the two
rational numbers is: %d/%d",
```

```
void equal(struct RATIONAL *p, struct RATIONAL *q)
{
   if((p->numerator)*(q->denominator)==(p-
   >denominator)*(q->numerator))
        {
        printf("\n The two rational numbers are equal.");
        }
   else
        printf("\n The two rational numbers are unequal.");
printf("\n\t*********equality check complete**********");
}
```

Array in C

- An array is a group of related data items that share a common name.
- The individual data items are called elements of the array and all of them are of same data type.
- The individual elements are characterized by array name followed by one, two or more subscripts (or indices) enclosed in square brackets.

  E.g. int num[30];
  - This statement tells the compiler that *num* is an array of type *int* and can store 30 integers.
  - The individual elements of *num* are recognized by *num[0]*, *num[1]*,…, *num[29]*.
  - The integer value within square bracket (i.e. [ ]) is called *subscript* or *index* of the array.
  - *Index* of an array always starts from 0 and ends with one less than the size of the array.
- The most important property of an array is that its elements are stored in contiguous memory locations.

Array as an ADT

- We can represent an array as an ADT by providing a set of specifications with a set of operations on the array with no regard to its implementation.
- The ADT **ARRAY** is a **finite** sequence of storage cells and is completely defined by its size and its data type with the condition that its size must be of type integer. The following operations are defined on it:
  - ***create(N, int):*** creates an array with storage for N items such that N is an integer
  - ***extract(a, i):*** returns the value of the item stored in the i^{th} position in the array **a** with 0<=i<N.
  - ***store(a, i, value):*** stores value in the i^{th} position of the array **a**.

Simulation --- --- --- Array as an ADT

```
#define ub 5
void extract(int a[ub], int i);
void store(int a[ub], int i, int elt);
void main()
{
int a[ub]={1,2,3,4,5};
int i;
int elt;
clrscr();
printf("\n Which array element you want to extract(Oth means first):");
scanf("%d", &i);
extract(a, i);
printf("\n What value you want to store at what position (0th means first))?");
printf("\n Value:");
scanf("%d", &elt);
printf("\n Position:");
scanf("%d", &i);
store(a, i, elt);
getch();
}
```

```
void extract(int a[ub], int i)
{
if(i<0 || i>ub-1)
   printf("\n Out of Bound error");
else
   printf("\n The value extracted at %dth position is %d", i, a[i]);
}


void store(int a[ub], int i, int elt)
{

if(i<0 || i>ub-1)
   printf("\n Out of Bound error");

else
   a[i] = elt;
   printf("\n The value stored at %dth position is %d", i, a[i]);
}
```

One dimensional array

- A list of items can be given one variable name using only one subscript (or dimension or index) and such a variable is called a *single-subscripted variable* or a *one-dimensional array*.
- The value of the single subscript or index from 0 to n-1 refers to the individual array elements; where n is the size of the array.
- E.g. the declaration *int a[5];* is a 1-D array of integer data type with 5 elements: *a[0]*, *a[1]*, *a[2]*, *a[3]*and *a[4]*.
- *Note: Generally a single **for** loop is used for input and output in a one-dimensional array.*

One dimensional array…

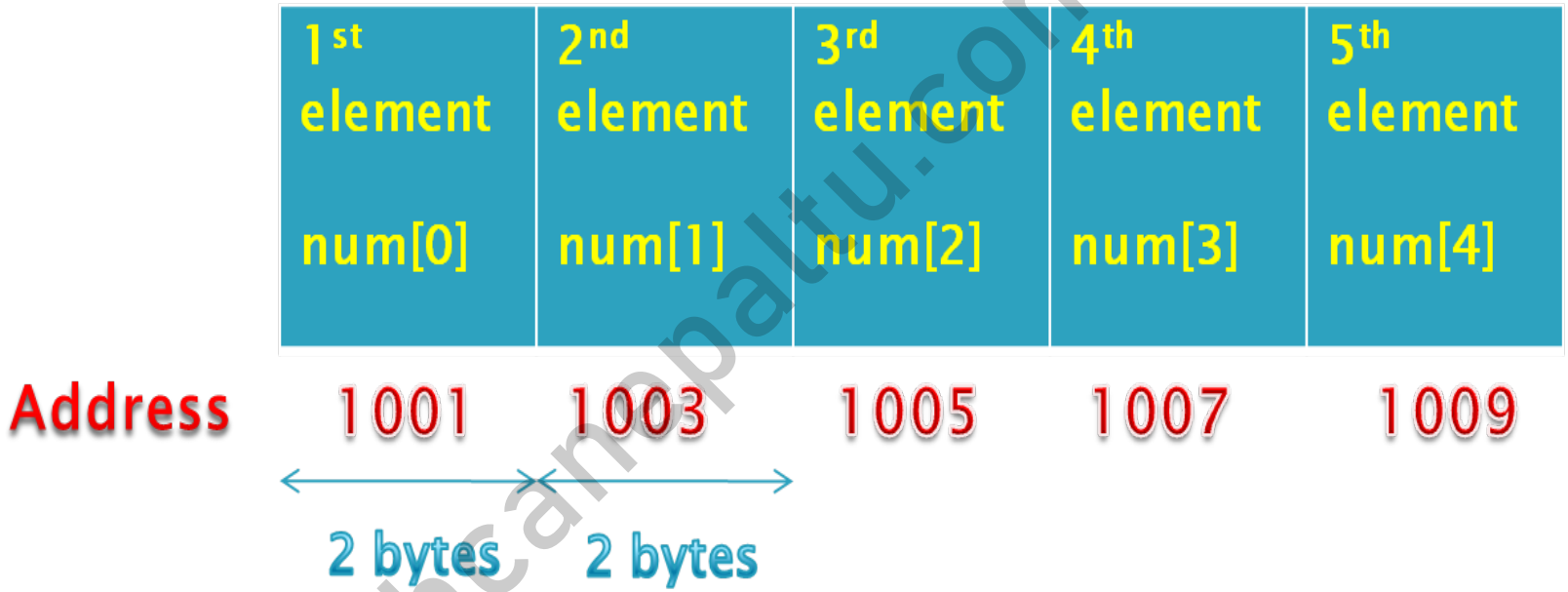


Fig: Memory Representation of a 1-D Array

How to declare array???

- Statically
- Two ways:

```
#define SIZE 100   void main()
void main()    {
{      int array[100];
int array[SIZE];    }
}
```

- Find out error here:

```
void main()
{
int n=10, i;
int marks[n];
clrscr();
for(i=0;i<n;i++)
printf("%d\t", marks[i]);
getch();
}
```

Size of an array must be either symbolic constant or integer constant.

Two Dimensional Array

- The two dimensional array is also called matrix.
  - Note: One dimensional array is also called vector.
- An m*n two dimensional array can be thought as a table of values having m rows and n columns.
- The syntax for 2-D array declaration is:

  data_type array_name[row_size][col_size];
- E.g.

  int matrix[2][3];

  Here, matrix is a 2-D array that can contain 2*3=6 elements. This array matrix can represent a table having 2 rows and 3 columns from matrix[0][0] to matrix[1][2].

Two Dimensional Array…

	Column1	Column2	Column3
Row1	matrix[0][0]	matrix[0][1]	matrix[0][2]
Row2	matrix[1][0]	matrix[1][1]	matrix[1][2]

OTE: ARRAY ELEMENTS ARE STORED ROW WISE IN MEMORY

Two Dimensional Array…

- Consider an array *marks* of size 4*3 with elements having values:
  int marks[4][3]={35,10,11,34,90,76,13,8,5,76,4,1};
- This array can be realized as a matrix having 4 rows and 3 columns as:

  35 10 11

  34 90 76

  13 8 5

  76 4 1
- To access a particular element of a 2-D array, we have to specify the array name, followed by two square brackets with row and column number inside it.
- Thus, marks[0][0] accesses 35, marks[1][1] accesses 90, marks[2][2] accesses 5 and so on.
- Note: Generally, a nested ***for*** loop is used for input and output in 2-D arrays.

Test

A 2-D array A[4][3] is stored row-wise in memory. The first element of the array is stored at location 80. Find the memory location of A[3][2] if each element of array requires 4 memory locations.

Multi Dimensional Array

- Multidimensional arrays are those having more than two dimensions and more than two pairs of square brackets are used to specify its dimensions.
- Example: A 3-D array requires three pairs of square brackets.
- Syntax for defining multidimensional array is:

  data_type array_name[dim1][dim2]…[dimN];

  Here, dim1, dim2,…,dimN are positive valued integer expressions that indicate the number of array elements associated with each subscript. Thus, total no. of elements=dim1*dim2*…*dimN

- E.g.

  int survey[3][5][12];

  Here, survey is a 3-D array that can contain 3*5*12=180 integer type data. This array survey may represent a survey data of rainfall during last three years (2009,2010,2011) from months Jan. to Dec. in five cities. Its individual elements are from survey[0][0][0] to survey[2][4][11].

- Note: Generally, more than two nested ***for*** loops are used for input and output in a multidimensional array.

TU Exam Question (2065)

- **What are the differences between two dimensional array and multidimensional array?**

Pointers

- **The & operator**

  The operator & (called “address of” operator) immediately preceding a variable returns the base address of the variable associated with it in memory.

Pointers…

- "A pointer is a variable that contains an address which is a location of another variable in memory"
- The data type of the pointer variable and data type of another variable whose address the pointer variable holds should be same.
- Declaration Syntax: *data_type *pointer_name;*

  This declaration tells the compiler three things about the variable *pointer_name*:

  1. The asterisk (*) tells that the variable *pointer_name* is a pointer variable
  2. *pointer_name* needs a memory location
  3. *pointer_name* holds the address of type *data_type*
- Example:

  *int *p;*
- This signifies that *p* is a pointer variable and it can store the address of an integer variable (The address of float variable cannot be stored in it).

Pointers…

- Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement such as:

  pointer_name = &variable_name;

  which causes the pointer_name to point to variable_name.

- Example: `int *p, num;`

  `p=&num;`

- Here, *p* contains address of *num* (or *p* points to *num*). This is called *pointer initialization.*

- Note: Before a *pointer* is initialized, it should not be used.

Pointers…

- **The * (star or asterisk) operator**
  - When * is used with variable declaration before a variable's name, it becomes a pointer variable, not a normal variable. E.g. int *p;
  - When * is used in front of pointer variable, it indirectly references ***the value at that address*** stored in the pointer. In this case * is also called *indirection* or *dereference* operator.

- **What is NULL pointer???**
  - A *null* pointer is a special pointer value that points nowhere or nothing.
  - The predefined constant *NULL* in stdio.h is used to define *null* pointer.
    - Example:

```
#define NULL 0
void main()
{
int *ptr=NULL;
if(ptr==NULL)
printf("Change NULL to 1 and there will be warnings!!!");
getch();
}
```

Structure

- A structure is a collection of data items of different data types under a single name.
- The individual data items are called members of the structure.
- The syntax for structure definition is:

```
struct structure_name
{
data_type member_variable1;
data_type member_variable2;
……..……;
data_type member_variableN;
};
```

- A structure definition creates a new data type that is used to declare structure variables. Once ***struct structure_name*** is declared as a new data type, then variables of that type can be declared as:

```
struct structure_name structure_variable;
```

- The member variables of a structure are accessed using the dot (.) operator.

Structure…

- The members of a structure variable can be initialized using the syntax:

  struct structure_name structure_variable={value1, value2, … , valueN};

  where, value1 is initialized to the first member, value2 to the second member and so on.

- **<u>Copying structure variables</u>**

  Two variables of the same structure type can be copied by using the assignment operator "=".

  E.g. ***structure_variable1= structure_variable2***

- Note: Two structure variables cannot be compared directly by using the comparison operators "==" and "!=". We may do so by comparing individual structure members.

Structure…

- **Array of Structure**

```
struct structure_name
{
data_type member_variable1;
data_type member_variable2;
……..……;
data_type member_variableN;
};
```

- An array of structure can be declared as:

```
struct structure_name structure_variable[100];
```

Structure…

- **Pointer to Structure**

```
struct structure_name
{
data_type member_variable1;
data_type member_variable2;
……..……;
data_type member_variableN;
};
```

- A structure type pointer variable can be declared as:

```
struct structure_name *structure_variable;
```

- However, this declaration for a pointer to structure does not allocate any memory for a structure but allocates only for a pointer, so that to access structure's members through pointer ***structure_variable***, we must allocate the memory using ***malloc()*** function.
- Now, individual structure members are accessed by using arrow operator -> as:

```
structure_variable->member_variable1
 structure_variable->member_variable2
… … … …
 structure_variable->member_variableN
```

Passing whole structure to function

- Whole structure can be passed to a function by the syntax:

  function_name(structure_variable_name);

- The called function has the form:

```
return_type function_name(struct structure_name structure_variable_name)
    {
    ... ... ... ... ...;
    }
```

- Note: In this call, only a copy of the structure is passed to the function, so that any changes done to the structure members are not reflected in the original structure.

Passing structure pointer to function

- In this case, address of structure variable is passed as an actual argument to a function.
- The corresponding formal argument must be a structure type pointer variable.
- Note: Any changes made to the members in the called function are directly reflected in the calling function.
- Syntax in calling function:

function_name(&structure_variable_name);

- Syntax in called function:

```
return_type function_name(struct structure_name * structure_variable_name)
{
... ... ... ... ...;
}
```

Passing array of structure to function

- The name of the array of structure is passed by the calling function which is the base address of the array of structure.
- Thus, any changes made to the array of structure by the called function are directly reflected in the original structure.
- Syntax in calling function:

  function_name(structure_variable_name);

- Syntax in called function:

return_type(struct structure_name structure_variable_name[])
{
… … … … …;
}

Union

- Unions are similar to structure in the sense that they are also used to group together data items of different data types.
- The distinction lies in the fact that all members within a union share the same memory space whereas each member within a structure is allocated a unique memory space.
- In case of union, the compiler allocates a memory space that is large enough to hold the largest variable type in the union.
- Since same memory space is shared by all the members of a union, only one variable can reside in the memory at a time, and when another variable is set in the memory, the previous variable is replaced by the new one.

Model Question (2008)

- **Explain the difference between structure and union.**

Self-Referential Structures

- A self-referential structure contains a pointer member that points to a structure of the same structure type.
- For example, the definition

```
struct node
{
int data;
struct node *nextPtr;
};
```

defines a type, *struct node.*

- A structure of type *struct node*, has two members: an *int* member *data* and another pointer member *nextPtr*.
- The member *nextPtr* points to (i.e. holds address of) another structure of type *struct node* i.e. a structure of the same type as the one declared here and hence is the term “self-referential structure”.

Self-Referential Structures…

- The member *nextPtr* is referred to as a link i.e. *nextPtr* can be used to "tie" a structure of type *struct node* to another structure of the same type.
- Self-referential structures can be linked together to form useful data structures such as lists, queues, stacks and trees.
- The following figure represents two self-referential structure objects linked together to form a list:

Self-Referential Structures…

- A NULL pointer is assigned to the *nextPtr* member of the last self referential structure to indicate that the *nextPtr* doesn't point to another structure.
- The NULL pointer indicates the end of a data structure just as the null pointer indicates the end of a string.
- Note: Not setting the *nextPtr* in the last node of a list to NULL can lead to runtime errors.

// An example of a linked list

```
#define NULL 0
struct node
   {
   int data;
   struct node *nextPtr;
   };
void main()
{
struct node node1,node2;
clrscr();
node1.data=100;
node1.nextPtr=&node2;
node2.data=200;
node2.nextPtr=NULL;
printf("\n");
printf("|%d|%u|-----> |%d|%u|",node1.data,node1.nextPtr,node2.data,node2.nextPtr);
printf("\n Address of node2=%u",&node2);
getch();
}
```

Unit 2: Algorithm

a. Concept and Definition
b. Design of algorithm
c. Characteristic of algorithm
d. Big O notation

Concept and definition

- "*An algorithm is a finite set of precise instructions executed in finite time for performing a computation or for solving a problem.*"
- To represent an algorithm we use ***pseudocode***.
- ***Pseudocode*** represents an algorithm in a clear understandable manner as like English language and gives the implementation view as like programming language.

Example: write an algorithm for finding the factorial of a given number.

```
fact(n)
{
fact=1;
for(i=1;i<=n; i++)
fact=fact*i;
return fact;
}
```

This algorithm is for getting factorial of *n*. The algorithm first assigns value 1 to the variable *fact* and then until *n* is reached, *fact* is assigned a value *fact*i* where value of *i* is from *1* to *n*. At last the value *fact* is returned. The pseudocode used here is loosely based on the programming language C.

Model question (2008)

- What are the major characteristics of algorithms?

Characteristics (properties) of algorithm

- **Input:** An algorithm has input values from a specified set.
- **Output:** From each set of input values, an algorithm produces output values from a specified set. The output values are the solution to the problem.
- **Definiteness:** The steps of an algorithm must be defined precisely.
- **Correctness:** An algorithm should produce the desired output after a finite (but perhaps large) number of steps for any input in the set.
- **Effectiveness:** It must be possible to perform each step of an algorithm exactly and in a finite amount of time.
- **Generality:** The devised algorithm should be capable of solving the problem of similar kind for all possible inputs.

- The above algorithm for factorial satisfies all the properties of algorithm as follows:
  - The input for the algorithm is any positive integer and output is the factorial of the given number.
  - Each step is clear since assignments, finite loop, the arithmetic operations and jump statements are defined precisely.
  - Each input gives its corresponding factorial value after a finite number of steps → Correctness.
  - When the return statement is reached, the algorithm terminates and each step in the algorithm terminates in finite time. All other steps, other than the loop are simple and require no explanation. In case of loop, when the value of i reaches $n+1$, then the loop terminates.
  - The algorithm is general since it can work out for every positive integer.

Complexity of algorithms

- When an algorithm is designed, it must be analyzed for its **efficiency** i.e. **complexity**.
- The **complexity** of an algorithm is defined in terms of **computational resource** needed by the algorithm.
- There are two kinds of ***computational resources*** used by an algorithm: CPU's processing power (**TIME COMPLEXITY**) and computer's primary memory (**SPACE COMPLEXITY**).

Complexity of algorithms...

- The measure of time required by an algorithm to run is given by **time complexity** and the measure of space (computer memory) required by an algorithm is given by **space complexity**.
- We focus on **time complexity**.
- Since actual time required may vary from computers to computers (e.g. on a supercomputer it may be million times faster than on a PC), we have to use the *number of steps required by the algorithm for a given set of inputs* to measure the **time complexity**.

- **Example (Time complexity):**
  - We can find the time complexity of the above algorithm by enumerating the number of steps required for the algorithm to obtain the factorial of *n*.
  - We can clearly see that the assignment of the variable *fact* is done once, the *for* loop executes n+1 times, the statement inside the *for* loop executes n times and the last operation that is the return statement executes single time.
  - Thus the total time in terms of number of steps required for finding the factorial of *n* can be written as:

    $f(n) = 1 + (n+1) + n + 1 = 2n+3.$

Big-Oh (O) notation

- The complexity of an algorithm is analyzed in terms of a mathematical function of the size of the input.
- The complexity analysis (**TIME**) of an algorithm is very hard if we try to analyze exact. So we need the concept of asymptotic notations.
- Big-O notation is one of the asymptotic notation used for complexity analysis of an algorithm.
- Definition: Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that *f(n)* is O(g(n)) (read as *f(n)* is big-oh of *g(n)*) if there are constants *C* and *k* such that

  $|f(n)| \leq C^*|g(n)|$

  whenever $n > k$.

- For example: In the above algorithm

$$f(n) = 2n+3 \leq 6.n$$

so that we can infer that $f(n) = O(g(n))$ with $g(n) = n$, $C=6$ and $n > 1$.

Hence the time complexity of the factorial algorithm is O(n) i.e. linear complexity.

Categories of algorithms

- Based on Big-O notation, the algorithms can be categorized as follows:
  - **Constant time algorithms → O(1)**
  - **Logarithmic time algorithms → O(log n)**
  - **Linear time algorithms → O(n)**
  - **Linearithmetic time algorithms → O(n log n)**
  - **Polynomial time algorithms → $O(n^k)$ for k>1**
  - **Exponential time algorithms → $O(k^n)$ for k>1**

Increasing order of complexity

Worst, Best and Average Case Complexity

- The *worst-case complexity* of an algorithm is defined as the maximum number of steps taken on any instance of size *n*.
- The *best-case complexity* of an algorithm is defined as the minimum number of steps taken on any instance of size *n*.
- The *average-case complexity* of an algorithm is defined as the average number of steps taken on any *random instance* of size *n*.

- Note: In case of running time, *worst-case time complexity* indicates the longest running time performed by an algorithm for any input of size *n*, and thus this guarantees that the algorithm finishes on time. Hence, algorithms are analyzed for their efficiency mostly in terms of its *worst-case complexity*.

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 Sequence
  - 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.
- *The 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…***PUSH***
- 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...***POP***
- Since items are inserted and deleted in this manner, a stack is also called a *last-in, first-out* (LIFO) list.
- *Note: Because of the push operation which adds elements to a stack, a stack is also called a* ***pushdown list****.*

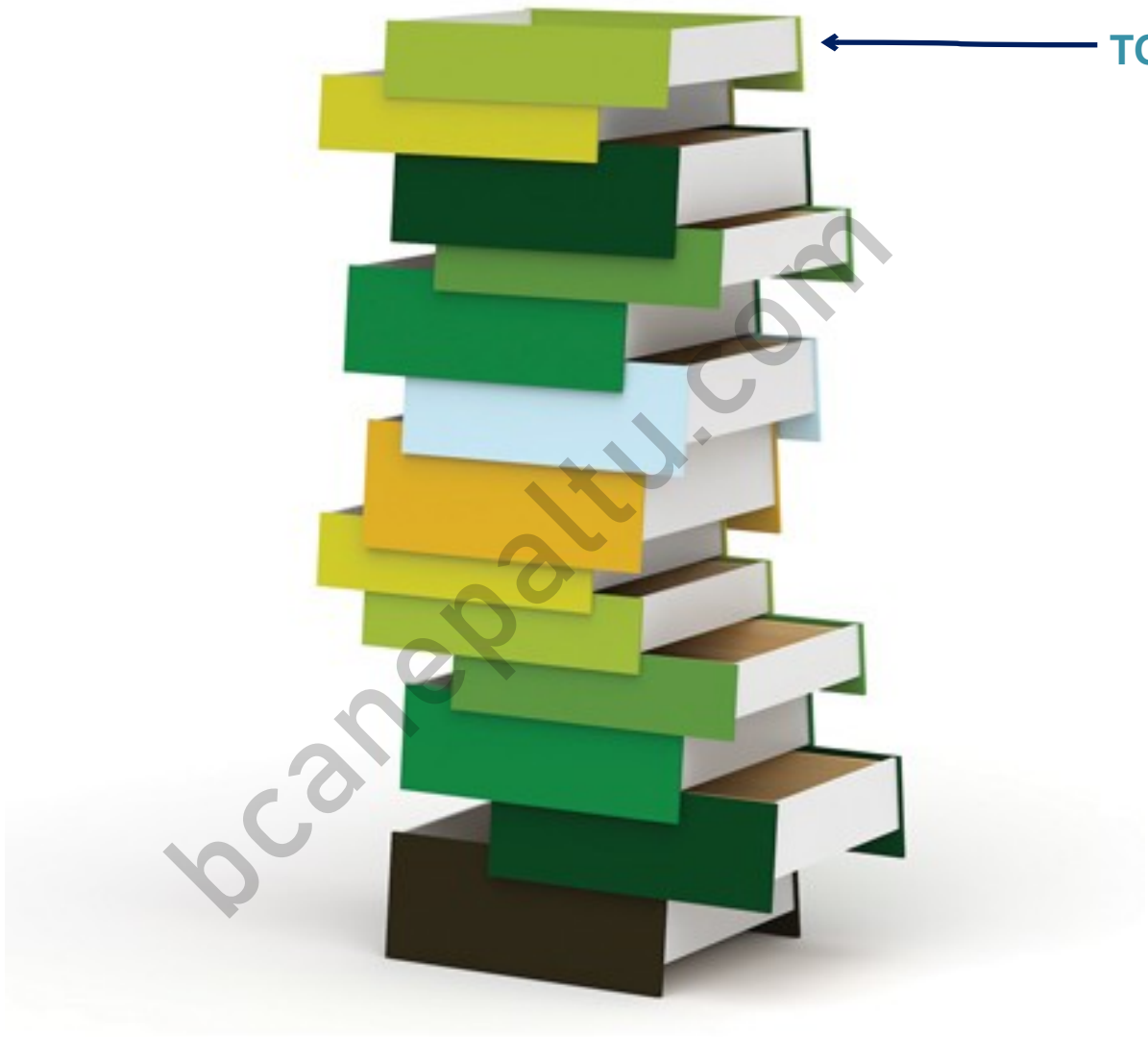


Fig: Stack

Primitive Operations on 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***.
- 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***.

- The operation ***empty(s)*** determines whether or not a stack ***s*** is empty. If the stack is empty, ***empty(s)*** returns ***TRUE***; otherwise it returns ***FALSE***.
- The operation ***stacktop(s)*** returns the top element of stack ***s***.
- *Note: 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

A **stack** is a linear collection of items, where an item to be added to the stack must be placed on **top** of the stack called **push** and items that are removed from the stack must be removed from the **top** called **pop**.

Operations

The operations on a stack are:

- **push(s, i)** - add an item **i** to the stack **s**.
- **pop(s)** – removes the top item from the stack **s**.
- **empty(s) –** check whether the stack **s** is empty.
- **stacktop(s)** - access item at the top of the stack **s** without removing it.

Representing stacks in C -- Array Implementation

- A stack in C is declared as a structure having 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 STACKSIZE 100
struct stack
{
int items[STACKSIZE];
int top;
};
```

- The size of the array is declared 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.
- Since stack top moves as items are pushed and popped, so to keep track of the current position of the top of the stack, the integer variable ***top*** is used.
- An actual stack can now be declared as:

```
struct stack s;
```

- To initialize the stack ***s*** to the empty state: ***s.top= -1***.
- To determine whether or not a stack is empty, test the condition: ***s.top==-1***.

Representing stacks in C…

- Note:
  - If *s.top = 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 becomes 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 is increased by 1 so s.top becomes 5 and the new item is inserted into s.items[5].*

Implementing the *pop* operation

- The function ***pop*** is implemented using the following three steps:
  1. If the stack is empty, print a warning message of *underflow* and halt execution.
  2. Remove the top element from the stack by returning this element to the calling program.
  3. Decrement the stack top.

```
//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

- In the array implementation of stack's *push* operation, 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, ***overflow*** occurs.
- The function ***push*** is implemented using the following four steps:
  1. If the stack is full, print a warning message of *overflow* and halt execution.
  2. Take value for the element that is to be pushed.
  3. Increment the stack top.
  4. Enter element into the stack top.

```
//C code to implement push
void push(struct stack *ps, int x)
{
if(ps->top == STACKSIZE-1)
   {
   printf("\n STACK OVERFLOW");
   exit(1);
   }
else
   {
   ++(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
    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 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 from ***left to right*** except in the case of ***exponentiation***, where the order is from ***right to left***.
- Thus A+B+C means (A+B)+C, whereas ABC 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

Classwork

- Convert the following expression to postfix and then evaluate the postfix expression:

  a + b - (c * d) / e

  where a=3, b=1, c=8, d=5 and e=2.

//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--]);
    }
```

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.

Converting an expression from Infix to Postfix

- Assumptions:

1. We scan one character at a time from left to right.
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 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. Otherwise ***pop*** operator to ***postfix*** string until the precedence of top of stack is greater than or equal to currently scanned operator and finally push currently scanned operator to ***opstack***.
5. Whenever right parentheses i.e. ')' is encountered, pop ***opstack*** until a matching left parentheses is found, add to postfix string and then cancel both parentheses.
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*	

Classwork

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.

```
#include <stdio.h>
#include <conio.h>
#include<string.h>
#include <ctype.h>
#include <stdlib.h>
#define STACKSIZE 100
struct opstack
      {
      char items[STACKSIZE];
      int top;
      };
void push(struct opstack *, char);
char pop(struct opstack *);
int checkprecedence(char);
void main(void)
{
char
      infix[STACKSIZE],postfix[STAC
      KSIZE],ch;
int i, j=0, l;
struct opstack s;
s.top=-1;
clrscr();
printf("\n Enter a valid infix:");
gets(infix);
l=strlen(infix);
for(i=0;i<l;i++)
{
      if(infix[i]=='(')
          push(&s,infix[i]);
      else if(isalpha(infix[i]))
          postfix[j++]=infix[i];
      else if(infix[i]==')')
      {
      while(s.items[s.top] != '(')
              postfix[j++]=pop(&s);
      ch=pop(&s);
      printf("\n %c is popped", ch);
      }
      else
      {
      if(s.top==-1)
              push(&s,infix[i]);
      else
      if(checkprecedence(s.items[s.top])
      <checkprecedence(infix[i]))
              push(&s,infix[i]);
      else
      {
       if(s.items[s.top]==infix[i] &&
      infix[i]=='$') // for associativity
                  push(&s,infix[i]);
       else
        {

      while(checkprecedence(s.items[s.to
      p])>=checkprecedence(infix[i]))
              postfix[j++]=pop(&s);
       push(&s,infix[i]);
       }
       }
      }
 }

while(s.top!=-1)
      postfix[j++]=pop(&s);

postfix[j]='\0';
printf("\n The postfix expression is:
      %s", postfix);
getch();
}
```

```
int checkprecedence(char p)
{
if(p=='$')
    return 4;
else if(p=='*' || p=='/')
    return 3;
else if(p=='+' || p=='-')
    return 2;
else
    return 1;
}

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

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

Converting an expression from Infix to Prefix

- 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, and
   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

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 arises any question 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.

Note:

- Prefix notation is also known as **Polish** notation while Postfix notation is also known as **Reverse Polish** notation.

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.
- 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 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;`
	`}`

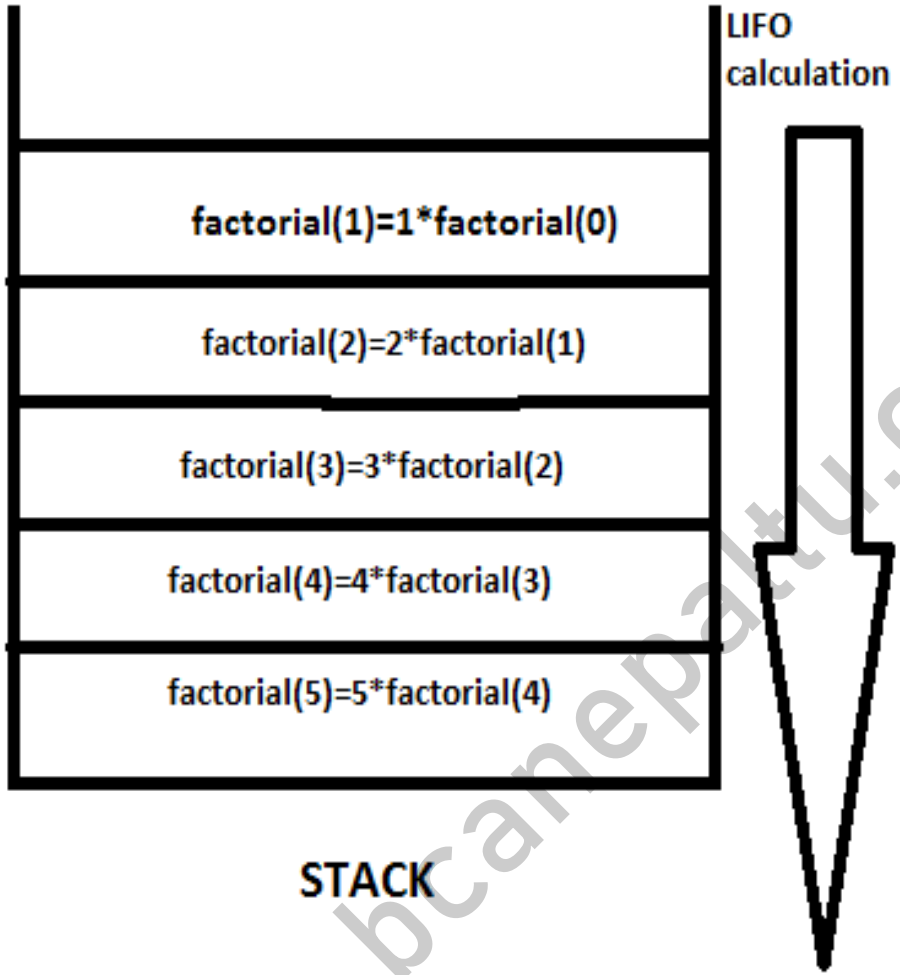
3. Problem to be solved is defined in terms of its previous result to solve a problem using recursion.	3. It is not necessary to define a problem in terms of its previous result to solve using iteration. For e.g. "Display your name 1000 times"
4. In recursion, a function calls to itself until some condition 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();
  }
```



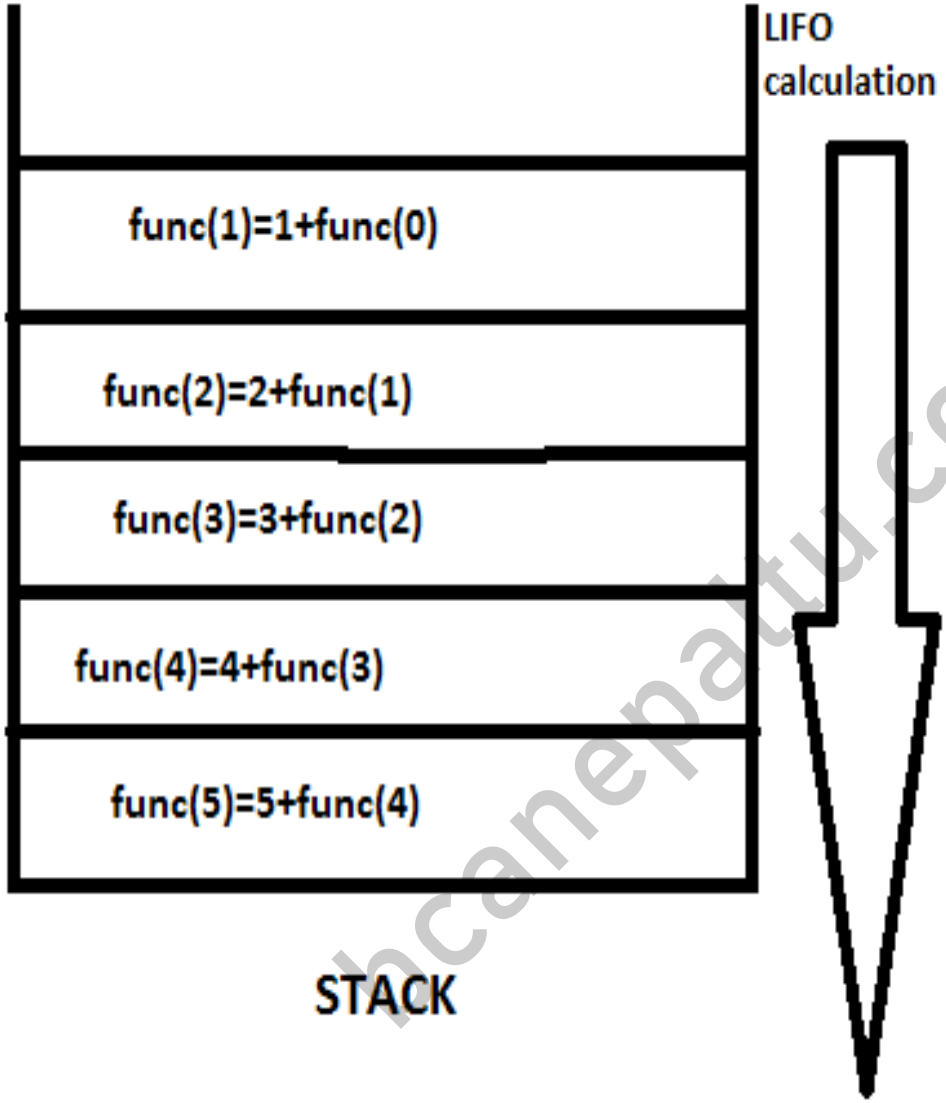
While calculating the factorial of n=5, the else part gets executed and the value 5*factorial(4) is pushed onto the stack. Then the factorial function gets called again recursively with n=4 and 4*factorial(3) is executed and is pushed onto the stack. This process goes on like this. When factorial(1) 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

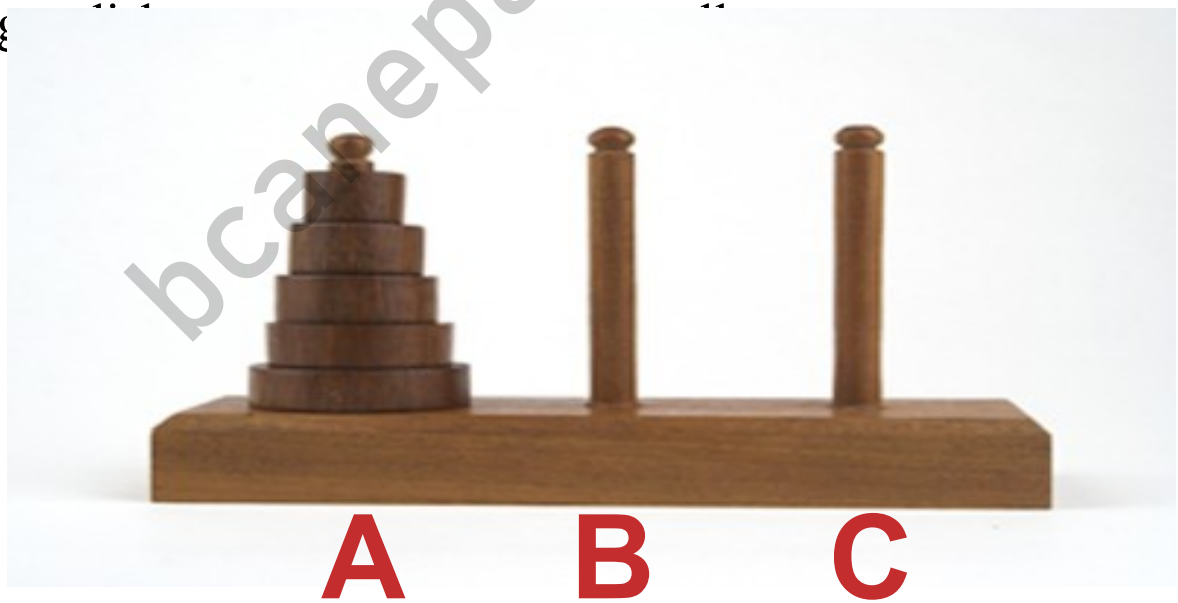
```
//The fibonacci sequence is: 1,1,2,3,5,8,13,…
//The following program computes the nth Fibonacci number
int fibo(int n)
{
if(n<=1)
return n;
else
return (fibo(n-1)+fibo(n-2));
}

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 larg



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.

- *Note: The number of moves required to solve a Tower of Hanoi puzzle is* $2^n - 1$*, where* ***n*** *is the number of disks.*

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);
  }
```

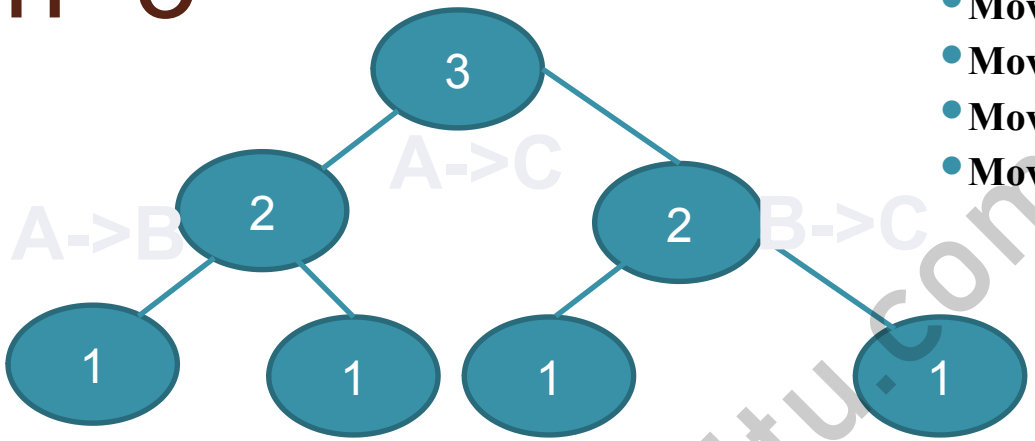
from: the peg from which we are removing disks

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, 'L', 'R', 'C');