Introduction To Data Structures

Data structure is the most fundamental subject in engineering. It addresses the issue of organizing the data in most efficient manner. The study of data structure involves two things the specification of data structure and its implementation. Specification tells what kind of data structure is required and implementation tells us how the data structure can be used.

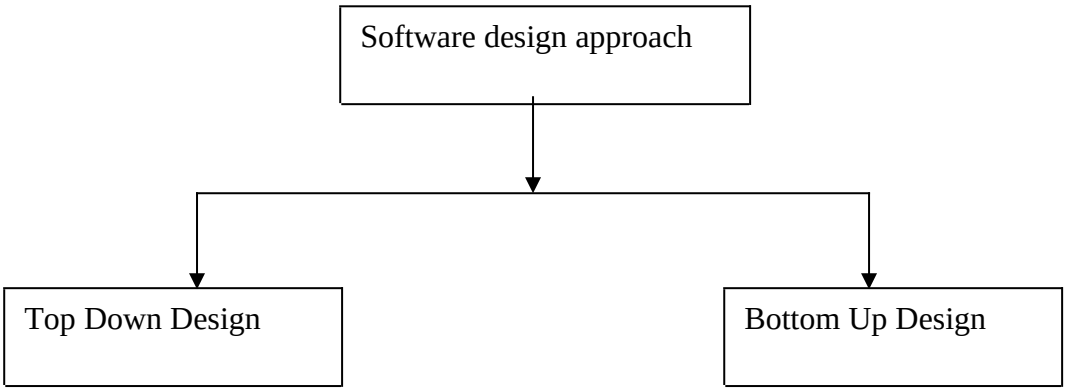
Unit I

Problem solving

In the computer science the big problem can be solved by building the algorithms. To solve any problem normally that problem is divided into sub problems and those sub problems are then solved. Then the solution of these sub problems is combined to obtain the solution of that main problem

There are two design approaches used in the problem-solving domain and those are

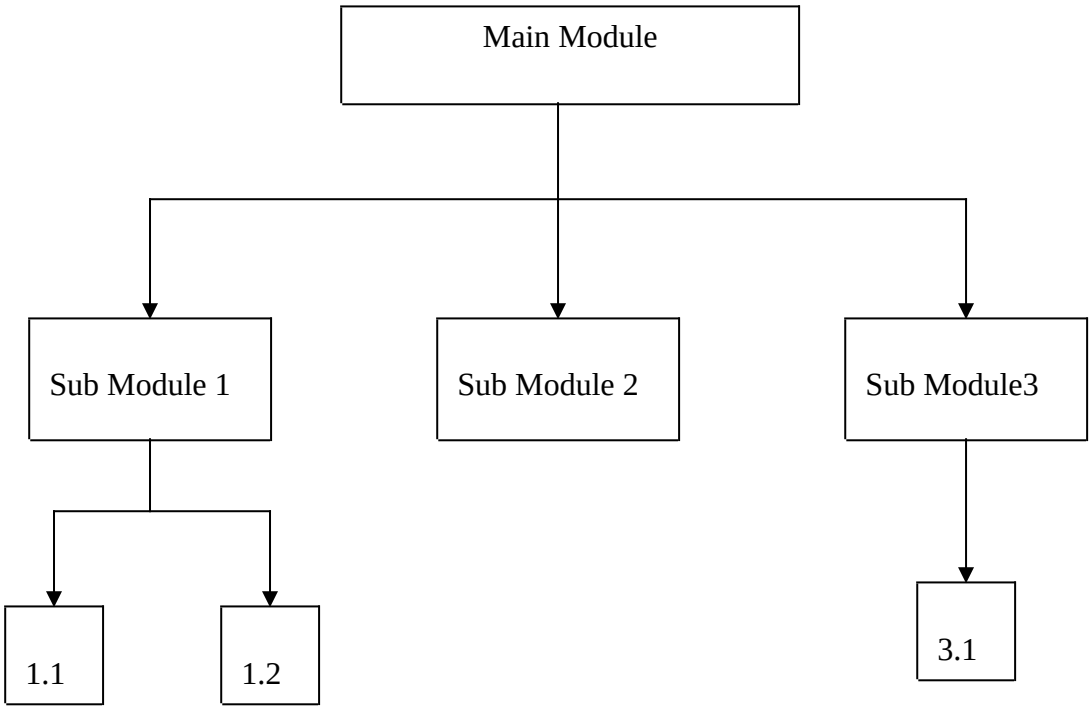
1. Top down design
2. Bottom up design



These software design strategies help a programmer to solve the complex problems efficiently.

Top Down Design:

In top down design method a big global problem is divided into sub problems and these sub problems then can be solved to get the global solution. Usually in high-level languages the top down design approach is used to solve any problem. The figure below shows the top down design approach.



For example if we want to find the sum of two numbers then we can solve this problem by using top down design method as follows.

/*program for implementation of top down design method. This program is for addition of two numbers*/

```
#include<stdio.h>
#include<conio.h>
int a,b,c,sum;
void getthetwo_num();
int addoftwo_num();
```

```
void print_result(int);
void main()
{
clrscr();
printf("\n addition of two num");
getthetwo_num();
addoftwo_num();
print_result(c);
getch();
}
void getthetwo_num()
{
printf("\n enter the value for a:");
scanf("%d “,&a)
printf(“\n enter the value for b:”);
scanf(“%d”,&b);
}
int addoftwo_num()
{
c=a+b;
return c;
}
void print_result(int c)
{
printf("the addition of two number is:%d",c);
}
```

output

```
 addition of two num
 enter the value for a:3
enter the value for b:3
the addition of two number is:6
```

In the above program we are first writing the main function which act as a main function module, this main function is to solve the problem of addition of two numbers.

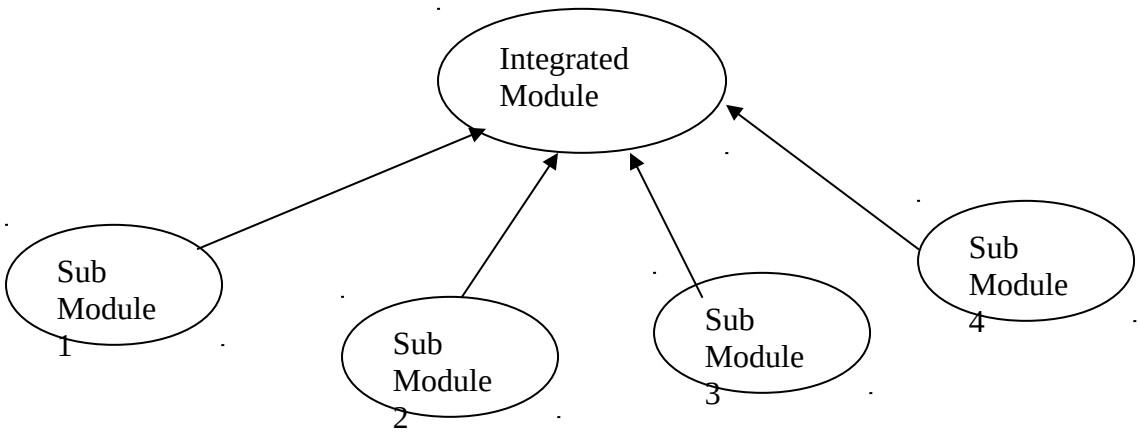
And to accomplish this task the sub tasks are

1. Accepting of two numbers
2. Performing two numbers
3. Printing the addition of two numbers
4. And for every sub task we have written functions. Thus the top down design approach is implemented.

Bottom Up Design:

In the bottom up design method the modules are first created and these modules then are integrated to form a solution of big module or problem.

The figure below represents the bottom up design approach.



For example if we want to find the sum of two numbers then we can solve this problem by using bottom up design method as follows.

```
/* this program is for implementing the bottom
up design for the program of addition of two numbers*/
#include<stdio.h>
#include<conio.h>
/*get_num function for accepting the two numbers*/
```

```
void get_num(int *x,int *y)
{
printf("\n enter the two numbers");
scanf("%d%d",x,y);
}
/* this sum function is for performing addition of two numbers*/
int sum(int a, int b)
{
int c;
c=a+b;
return c;
}
/* this print_ans function is for printing the addition as aresult*/
void print_ans(int z)
{
printf("the addition of two number=%d",z);
}
/*integrated module can be given by main function*/
void main()
{
int a,b,c;
clrscr();
printf("\n program for addition of two numbers:");
get_num(&a,&b);
c=sum(a,b);
print_ans(c);
getch();
}
```

output:

```
 program for addition of two numbers:
 enter the two numbers6
```

6

The addition of two number=12

Algorithmic analysis:

An algorithm is a finite set of instructions for performing a particular task. The instructions are nothing but the statement in simple English language.

Example:

Let us take a very simple example of an algorithm which adds the two numbers and the result in a third variable.

Step1: Start.

Step2: read the first number in variable ‘a’

Step3: Read the second number in variable b

Step4: perform the addition of both the numbers and store the result in variable ‘c’

Step5: print the vale of ‘c’ as a result of addition.

Step6: Stop.

Characteristics of algorithm:

1. Each algorithm is supplied with zero are external quantities. Here supplying external quantities means giving input to the algorithm.
2. Each algorithm must produce at least one quantity. Production of one quantity means there should be some output.
3. Each algorithm should have definiteness i.e. each instruction must be clear and unambiguous.
4. Each algorithm should have finiteness i.e. if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after finite number of steps.
5. Each algorithm should have effectiveness.

We need to distinguish between an algorithm and a program. The major difference is that algorithm always terminates while a program may not always terminate.

The efficiency of algorithm is directly related to the efficiency of program.

There are many criteria upon which we can judge program. For instance

i) Does my program work on what I want to do?

ii) Does it work correctly according to original specifications?

iii) Is the code readable?

The above criteria are important when one writes the software. Still there are other criteria for judging programs, which have a more direct relationship with performance. These are mainly concerned with computing time and storage requirements of the algorithms.

Suppose we want to find the time taken by following program statement.

X=x+1

If one ask the question that how much time it takes to execute this statement. It is impossible to determine exact time taken by the statement to execute unless we have the following information.

i) The machine that is used to execute this statement,

ii) Machine language instruction set.

iii) The time required by each machine instruction.

iv) Compilation time (time taken by the compiler to make this statement to machine language)

v) And the kind of operating system (multi programming or time sharing)

Although, finding out the above-mentioned information is not difficult but this is not attractive option. The information may vary to machine to machine and perhaps we won't get the exact figures. So the better idea to determine the time taken by each instruction to execute is the frequency count.

Frequency count:

For example

1) x=x+1

As this statement executes only once therefore the frequency count for this statement is 1.

2) for(I=0;I<n;I++) - this loop will be executed n times

   x=x+1; - this statement will be executed 1 time

   Above statement execute n times therefore the frequency count will be n. thus the performance of a program is measured in terms of "time complexity" i.e. total time taken by the algorithm to execute.

Let us see more program fragments and have more discussion on frequency count.

Example 1:

```
Main()
{
int j,n,a=10;
printf("\n Enter the value of n");
scanf("%d",&n);
for(j=0;j<n;j++)
a++;

}
```

Now let us analyse the above program fragment –

Concentrate on 'for' loop

i) j=0 will be executed one time

ii) j<n will be executed n time i.e.

iii) j++ will be executed n times

iv) a++ will be executed n times.

Totally 1+(n+1)+n+n i.e. 3n+2

Generally when we sum the frequency count of all the statements in the given program we get a polynomial. However, for the sake of analysis we are interested in the order of magnitude of polynomial.

So we will drop the constant terms from the polynomial. Hence the order of magnitude is 'n' and it is denoted as O(n). "O" is called big-oh notation.

Example 2:

```
Main()
{
        int j,k,n,a=10;
        printf("\n Enter the value of n");
        scanf("%d",&n);
        for(I=0;I<n;I++)
        {
```

```
            for(k=0;k<n;k++)
            {
                a++;
            }
        }
    }
```

1. j=0 will be executed once.
2. the statement j<n will be executed n+1 times.
3. the frequency count for inner ‘for’ loop will be :

   k=0 once

   k<n n+1 times

   k++ n times

   a++ n times

   3n+2 i.e. n times
4. j will be incremented n times and the inner loop will be executed n times
   a. thus the total frequency count will be n*n i.e. n^2 times. There fore the frequency count for the above program will be n^2 and will be denoted as $O(n^2)$.

How to choose best algorithm?

If we have two algorithms that perform same task, and the first one has a computing time of O(n) and the second of $O(n^2)$, then we will usually prefer the first one.

The reason for this is that as n increases the time required for the execution of second algorithm will get far more than the time required for the execution of first. We will study various for computing function for the constant values. The graph given below will indicate the rate of growth of common computing time functions.

N	Log_2n	$N \log_2n$	N^2	N^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536
32	5	160	1024	32768	4294967296

Notice how the times O(n) and O(nlogn) grow much more slowly than the others. For large data sets algorithms with a complexity greater than O(nlogn) are often impractical. The very slow algorithm will be the one having the time complexity 2^n.

Asymptotic notations

There are basically three types of mathematical notations for three different cases of time complexities.

1. Big oh or oh notation denoted as 'O'.
2. Omega notation denoted as Ω.
3. Theta notation denoted as θ.

Let us see one by one each notation.

1. Definition of big-oh notation:

   F(n) = O(g(n))

This means that always O(n) will give maximum amounts of time algorithm needs. I.e. O(n) gives "upper bound" on the time complexity value for the algorithm.

2. Definition of omega notation:

   f(n) = Ω(g(n))

This means that always Ω(n) will give minimum amount of time the algorithm needs i.e. it gives "lower bound" on the time complexity value for the algorithm.

3. Definition of theta notation:

   f(n) = θ(g(n))

This means that both upper bound and lower bound on f(n). I.e. worst and best cases require the same amount of time to within constant factor.

Examples of asymptotic notations:

1. **Big-oh and omega notation:**

Consider, an algorithm to search for a specific number from a set of n numbers we get two different time complexities.

One is for the best case and other is for the worst case. The best case for the algorithm is when the element to be searched is the one, which is stored at very first location. The worst case is when the number to be searched is placed at the end of the list. Thus in the best case only one comparison is needed while as in the worst-case n comparisons are needed. The best case as the time complexity as $\Omega(n)=1$ while the time complexity in the worst case will be $O(n)=n$.

2. Theta notation:

This indicates the lower and upper bound frequency. As an example consider an algorithm for finding out smallest number from a set of n numbers. Clearly this algorithm has identical for best and worst case time complexities. To find the smallest number from given list the maximum time required to scan the list and the minimum time required to scan the list will be the same. Clearly the time complexity of this algorithm is both $O(n)$ and $\Omega(n)$ so we now represents the time complexity of this algorithm as $\theta(n)$.

Space complexity:

Another useful measure of an algorithm is the amount of storage space it needs. The space complexity of an algorithm can be computed by considering the data and their sizes. Again we are concerned with those data items, which demand for maximum storage space. A similar notation 'O' is used to denote the space complexity of an algorithm. When computing for storage requirement we assume each data element needs one unit of storage space. While as the aggregate data elements in an array this assumption again is independent of the machines on which the algorithms are to be executed.