Total No. of Questions – [4]

Total No. of Printed Pages: 01

U218-124 (T1)

OCTOBER 2018/IN-SEM (T1)

S. Y. B. TECH.(COMPUTER ENGINEEERING) (SEMESTER - I)

COURSE NAME: FUNDAMENTALS OF DATA STRUCTURE

COURSE CODE: CSUA21174

(PATTERN 2017)

Time: [**1Hour**] [Max. Marks: **30**]

Marking Scheme

Q.1)a) Explanation of static data structure with example:	3M
Explanation of dynamic data structure with example:	3M
b) Explanation of ADT:	3M
Natural Number ADT:	3M
c) Explanation of Big-O notation:	2M
Frequency count example:	2M
OR	
Q.2) a) Big- Omega notation:	2M
Theta notation:	2M
Big-O notation:	2M
b) Explanation of each of 6 statement. 6 Steps* 1 M.	6M
c) Explanation of Recursion:	1M
Example:	3M
Q.3) a) Sparse Description:	2M
Example of Sparse:	1M
Algorithm for simple transpose:	3M
b) Address calculation of Row major with example:	2M
Address calculation of Column major with example:	2M
c) Reverse string C++ code using pointers:	4M.
OR	
Q.4) a) Explanation of representation of polynomials:	4M
Example:	2M
b) C++ code for string concatenation:	4M
c) Any two differences:	2M
Two examples of ordered list:	2M

Total No. of Questions – [4]

Total No. of Printed Pages: 12

U218-124 (T1)

OCTOBER 2018/IN-SEM (T1)

S. Y. B. TECH. (COMPUTER ENGINEERING) (SEMESTER - I)

COURSE NAME: FUNDAMENTALS OF DATA STRUCTURES

COURSE CODE: CSUA21174

(PATTERN 2017)

Time: [**1Hour**] [Max. Marks: **30**]

SOLUTION: FUNDAMENTALS OF DATA STRUCTURES

Q.1) a) Explain static and dynamic data structure with suitable example. [6 marks]

➢ **Static Data Structure:**

In Static data structures, memory for variables is allocated when the program starts. The size is fixed when the program is created. It applies to global variables, file scope variables, and variables qualified with static defined inside functions.

- In case of static data structure, m/m for objects is allocated at the time of compilation.
- Amount of m/m required is determined by the compiler during compilation.
- Ex. Arrays int a[50];

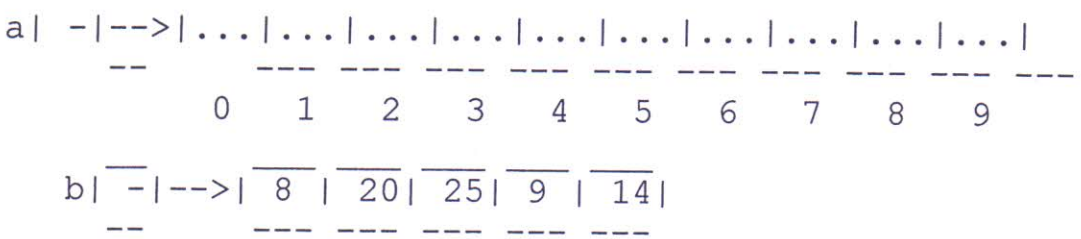
Statically declared arrays are allocated memory at compile time and their size is fixed, i.e., cannot be changed later

For example two int arrays are declared, one initialized, one not.

int a[10];

int b[5] ={8, 20, 25, 9, 14};

The array has 10 elements with subscripts numbered from 0 to 9, filled with garbage. The b array has 5 elements with subscripts numbered from 0 to 4, filled with the five given values. They accessed in the usual way, e.g., a[5] or b[2]. The memory is allocated at compile time. A memory picture of these arrays shows 10 int garbage values and 5 valid int values:



➢ Disadvantages:

- Wastage of m/m.
- It may cause overflow.
- No re-usability of allocated m/m.
- Difficult to guess exact size of data at time of writing of program.

➢ **Dynamic Data Structure:**

- Here m/m space required by variables is calculated & allocated during execution.
- Dynamic m/m is managed in 'C' and 'C++' through set of library functions.
- malloc & calloc functions are used in 'C' and new and delete operator is used for dynamic m/m allocation in C++.
- Linked data structures are preferably implemented using dynamic data structures.
- It gives flexibility of adding, deleting or re arranging data objects at run time.
- Additional space can allocated at run time.
- Unwanted space can be released at run time.
- It gives reusability of m/m space.

Example of dynamic memory allocation using malloc in 'C':

```
#include <alloc.h>
#include <process.h>
int main()
{
  char *str;
  /* allocate memory for string */
  if ((str = (char *) malloc(10)) == NULL)
  {
    printf("Not enough memory to allocate buffer\n");
    exit(1);  /* terminate program if out of memory */
  }
  /* copy "Hello" into string */
  strcpy(str, "Hello");
  /* display string */
  printf("String is %s\n", str);
  /* free memory */
  free(str);
  return 0;
}
```

Or

Example of dynamic memory allocation using 'C++':

Any Example containing new and delete operator. Ex.

```
node *delete_begin(node *first)
{
      node *a;
      a= new node;
      cout<<"Enter Employee name:";
      cin>>a->name;
      cout<<"Enter Department of Employee    :";
      cin>>a->dpt;
      cout<<"Enter Employee Id    :";
      cin>>a->empid;
      cout<<"Enter Salary of Employee :";
      cin>>a->sal;
      a->next=NULL;
```

```
        if(first==NULL)
        {
              cout<<"\n List is Empty";
        }
        else
        {
              a=first;
              first=first->next;
              cout<<"\n Deleted record is: \n %s \t %s \t %d \t %d \n",a-
>name,a->dpt,a->empid,a->sal;
              delete a;
        }
        return first;
}
```

b) What is Abstract data type? Explain an ADT for Natural Number. [6 marks]

Abstract data type:

- ADT it is a triple of
  - D-set of domain
  - F-set of function
  - A-Axioms in which only what is to be done is mentioned but how to be done is not mentioned.

In ADT, all the implementation details are hidden. So
ADT=Type+ Function name+ Behavior of each function

Explanation of ADT:

- A big program is broken down in smaller modules.
- Each module is developed independently.
- When the program is hierarchical organized it utilizes services of functions which utilizes services of other functions without knowing their implementation details.
- This is called as abstraction.
- Ex. int x,y,z;
- X=13;(details of storage is hide)
- Z=x+y; (details of + is hide)

Advantages of ADT:

Avoids redundancy of code Ex: Simulate waiting line of a bank.
Approach1: program that simulates bank queue. It cannot be reused for simulation of any other queue
Approach2: design queue ADT to solve any queue problem. Place it in library for all programmers to use.

- We don't need to know how a car (or a fridge) works in order to use one!
- all you need to know is what **operations** it supports and how to use those operations
- *abstract data types* (ADTs) are a collection of data (values) and all the operations on that data

Explain an ADT for Natural Number:

ADT *NaturalNumber*
Objects: Ordered subrange of integer from 0 to (INT_MAX)
Functions:

x,y= *NaturalNumber*, True, False= *Boolean*
NaturalNumber Zero() :=0
Boolean IsZero(x) := if(x) return False
else return True
Boolean Equal(x,y) := if(x==y) return True
else return False
Similarly write Successor(x), Add(x,y), Subtract(x,y), Predecessor(x,y), IsGreater(x,y), Multiply(x,y), Divide(x,y) etc.

c) Explain Big-O notation with suitable example. [4 marks]

Big - O notation:

- Worst case: Algorithm will give its worst behavior if the element to be searched is the last element in array or search ends in a failure.
- n comparison will be needed to search an element.
- Big - O notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.
- That means Big - O notation always indicates the maximum time required by an algorithm for all input values.
- That means Big - O notation describes the worst case of an algorithm time complexity.
- worst case= O(n)

Big - Oh Notation can be defined as follows.

- Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as O(g(n)).
- f(n) = O(g(n))

Example to calculate Big-O Time complexity:

i. Calculate frequency count:

```
#include<stdio.h>
void main()
{
   int i,n,sum,x;
1  sum=0;
2  printf("enter no. of data to be added");
3  scanf("%d",&n);
4  for(i=1;i<=n;i++)
   {
5     scanf("%d",&x);
6     sum=sum+x;
   }
7     printf("sum= %d",sum);
}
```

- Calculation of computation time:

Statement no.	Frequency	computation time
1	1	t1
2	1	t2
3	1	t3
4	n+1	(n+1) t4
5	n	nt5
6	n	nt6
7	1	t7

Total computation time= t1 + t2 + t3 + n(t4 +t5+ t6) + t4 + t7
T=n(t4 + t5+ t6) +(t1+t2+t3+t4+t7)
For large n T can be approximated to
T=n(t4+t5+t6) = kn
For faster computers time required for execution of (t1+t2+t3+t4+t7) is less.

Time complexity is Big-Oh of n.

OR

Q.2) a) Explain asymptotic notations. [6 marks]

Asymptotic Notations are allows us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases. This is also known as an algorithm's growth rate.

There are 3 asymptotic notations.

i. Big – Omega
ii. Big – O
iii. Big – Theta

i. Big – Omega:

- Best case: Algorithm will give its best behavior if the element to be searched is the first element in array.
- Only one comparison will be needed to search an element.
- Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.
- That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values.
- That means Big - Omega notation describes the best case of an algorithm time complexity.
  Best case= **Ω** (1)
  Big - Omega Notation can be defined as follows...
  - Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C x g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as Ω(g(n)).
  - f(n) = Ω(g(n))

ii. Big – O :

- Worst case: Algorithm will give it's worst behavior if the element to be searched is the last element in array or search ends in a failure.
- n comparison will be needed to search an element.
- Big - O notation is used to define the upper bound of an algorithm in terms of Time Complexity.
- That means Big - O notation always indicates the maximum time required by an algorithm for all input values.
- That means Big - O notation describes the worst case of an algorithm time complexity.
- worst case= O(n)

Big - Oh Notation can be defined as follows...

- Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as O(g(n)).
- f(n) = O(g(n))

iii. Big – Theta:

- Number of comparisons required to search an element present in between 1 and n.
- Big - Theta notation is used to define the average bound of an algorithm in terms of Time Complexity.
- That means Big - Theta notation always indicates the average time required by an algorithm for all input values.
- That means Big - Theta notation describes the average case of an algorithm time complexity.
- Average case= Θ (n)

Big - Theta Notation can be defined as follows...

- Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If C_1 g(n) <= f(n) >= C_2 g(n) for all n >= n_0, C_1, C_2 > 0 and n_0 >= 1. Then we can represent f(n) as Θ(g(n)).
- f(n) = Θ(g(n))

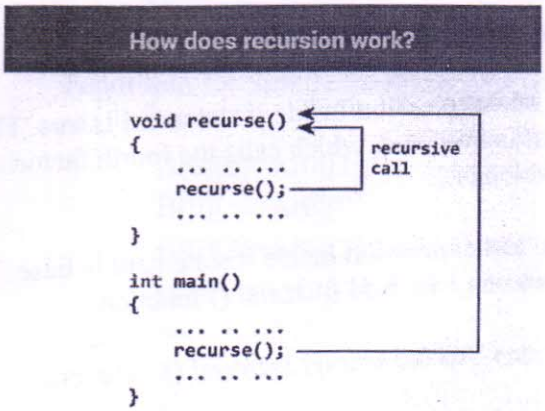
b) Describe the following statements. i) int a, *b=&a ii) int p, *p iii)a=(float*)&x iv) int **q v) char *s vi) int (*p)++ [6 marks]

1 mark each i) int a, *b=&a : pointer b will point variable a ii) int p, *p: multiple declaration of variables is not allowed iii)a=(float*)&x: Address of x will be type casted to an address of floating point number and this address is assigned to a. Since a pointer cannot be assigned to a variable, it will give an error iv) q is a double pointer. It can store the address of a single pointer) char*s: pointer to character vi) int (*p)++: Such declaration is not allowed

c) What is recursion? Explain with suitable example? [4 marks]

A function that calls itself is known as recursive function. And, this technique is known as recursion.

The figure below shows how recursion works by calling itself over and over again.



The recursion continues until some condition is met. To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call and other doesn't.

Example of Recursion:

```
#include <iostream>
using namespace std;
int factorial(int);
int main()
{
   int n;
   cout<<"Enter a number to find factorial: ";
   cin >> n;
   cout << "Factorial of " << n <<" = " << factorial(n);
   return 0;
}
int factorial(int n)
{
   if (n > 1)
   {
      return n*factorial(n-1);
   }
   else
   {
      return 1;
   }
}
```

Suppose the user entered 4, which is passed to the factorial () function.

1. In the first factorial () function, test expression inside if statement is true. The return num*factorial (num-1); statement is executed, which calls the second factorial () function and argument passed is num-1which is 3.

2. In the second factorial () function, test expression inside if statement is true. The return num*factorial (num-1); statement is executed, which calls the third factorial () function

and argument passed is num-1 which is 2.

3. In the third factorial () function, test expression inside if statement is true. The return num*factorial (num-1); statement is executed, which calls the fourth factorial () function and argument passed is num-1 which is 1.

4. In the fourth factorial () function, test expression inside if statement is false. The return 1; statement is executed, which returns 1 to third factorial () function.

5. The third factorial () function returns 2 to the second factorial () function.

6. The second factorial () function returns 6 to the first factorial () function.

7. Finally, the first factorial () function returns 24 to the main () function, which is displayed on the screen.

Q.3) a) Explain sparse matrix with suitable example? Write an algorithm for Simple transpose of sparse matrix [6 marks]

Sparse matrix:

Sparse matrix is that matrix which has a very few non zero elements.

- An example sparse matrix:

$$A = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

- A lot of "zero" entries. Thus large memory space is wasted in normal matrix.
- Sparse representation is used to save memory space.
- Uses triple <row, col, and value> to characterize an element in the matrix.
- Use array of triples a[] to represent a matrix.

	Row	Col.	Value
a[0]	6	6	8
a[1]	0	0	15
a[2]	0	3	22
a[3]	0	5	-15
a[4]	1	1	11
a[5]	1	2	3
a[6]	2	3	-6
a[7]	4	0	91
a[8]	5	2	28

Algorithm for Simple transpose of sparse matrix:

```
B[0][0]=A[0][1];
B[0][1]=A[0][0];
B[0][2]=A[0][2];
noterms=A[0][2];
noc=A[0][1];
if(A[0][2]>1)
{
nxt=1;
 for(c=0;c<noc;c++)//loop till col number of nonzero elements
 {      for(Term=1;Term<=noterms;Term++)// loop till we have nonzero
                                        elements
        {
/* if a column number of current triple == c, then insert the current triple in B
*/

if(A[Term][1]== c)
        {
                B[nxt][0]=A[Term][1];
                B[nxt][1]=A[Term][0];
                B[nxt][2]=A[Term][2];
                nxt++;
        }
  }
 }
}
```

- Complexity of simple transpose is O(No.of columns* No. of terms)

b) Derive a formula to access an element in the i^{th} row and j^{th} column of a matrix stored in row major form. Explain with example. [4 marks]

- The elements in two dimensional array may be arranged either in row wise or column wise.
- If the elements are stored in row wise manner then it is called "Row Major Representation".
- If the elements are stored in column wise manner then it is called "Column Major Representation".

Row major Representation:

➢ If the elements are stored in row wise manner then it is called "Row Major Representation".

- Ex. If we want to store elements
  - 10 20 30 40 50 60 then

In a 2D array it is represented as

0 1 2

0	10	20	30
1	40	50	60
.			
.			
.			
9			

- Each element is occupied at successive location if the element is of integer type then 2 bytes of memory will be allocated.
- Ex. int a[3][2]={ {10,20}
  {30,40}
  {50,60} }
- Then in a row major matrix it is stored at addresses as shown. Base address is assumed 100.

a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a [2][1]
100	102	104	106	108	110

So formula for address calculation for any element will be derived as follows.
In row major matrix, the element a[i][j] will be =(base address + i * total number of columns + j)* Size of data type.

c) Write C++ function to reverse a string using pointers [4 marks]

```
void reverse(char *a)
{
        int len=length(a);
        char temp;
        int j=len-1;
        for(int i=0;i<(len/2);i++)
        {
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
                j--;
        }
}
```

OR

Q.4) a) Explain representation of polynomial using array with suitable example. [6 marks]

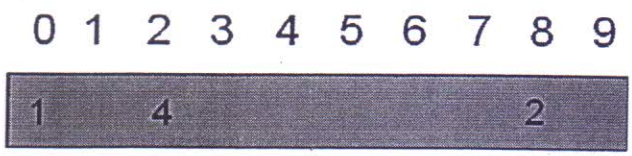
Polynomials:

- One classic example of an ordered list is polynomials.
  - Def. - A polynomial is the sum of terms where each terms consists of variable, coefficient and exponent.

Polynomial representation using one dimensional array:

one dimensional array where:
index represents the exponent
and the stored value is the
corresponding coefficient

$2x^8 + 4x^2 + 1$



Polynomial representation using arrays of structures:

```
// an array of struct
#define MAXSIZE 10
typedef struct poly {
                      real coeff;
                      int  expo;
                     }term;
term  poly1[MAXSIZE];
term  poly2[MAXSIZE];
term  poly3[MAXSIZE];
```

b) Write C++ code for concatenation of two strings without using library functions. [4 marks]

```
void concat(char *a,char *b)
{
        int i=0;
        while(a[i]!='\0')
                i++;
        for(int j=0;b[j]!='\0';j++)
        {
                a[i]=b[j];
                i++;
        }
        a[i]='\0';
}
```

c) Compare Array & Ordered list. Write two examples of ordered list. [4 marks]

Difference between array and ordered list (Any 2 differences):

i. A list is a different kind of data structure from an array.

ii. The biggest difference is in the idea of direct access Vs sequential access. Arrays allow both; direct and sequential access, while lists allow only sequential access. And this is because the way that these data structures are stored in memory.

iii. In addition, the structure of the list doesn't support numeric index like an array is. And, the elements don't need to be allocated next to each other in the memory like an array is.

iv. An array is a contiguous segment of memory, and a list is just a bunch of nodes, each one has the pointers to the "next" node (and also to the "previous" node, in the case of bidirectional lists).

Two examples of ordered list are (Any 2 example):

i. days of week:
daysOfWeek = {S,M,T,W,Th,F,Sa}

ii. digits:
integer = {0, +1, -1, +2, -2, +3, -3, …}

iii. Month of the year:
MonthOfYear = {JAN,FEB,MAR,APR,MAY,JUN,….}