Internal Assessment Test 1 – Nov 2024

Solution

Sub	Data Structures and Applications					Sub code	BCS304
Date	07/11/24	Duration	**90 mins**	Max Marks	**50**	Sem /Sec	III A, B&C

Answer any FIVE FULL Questions

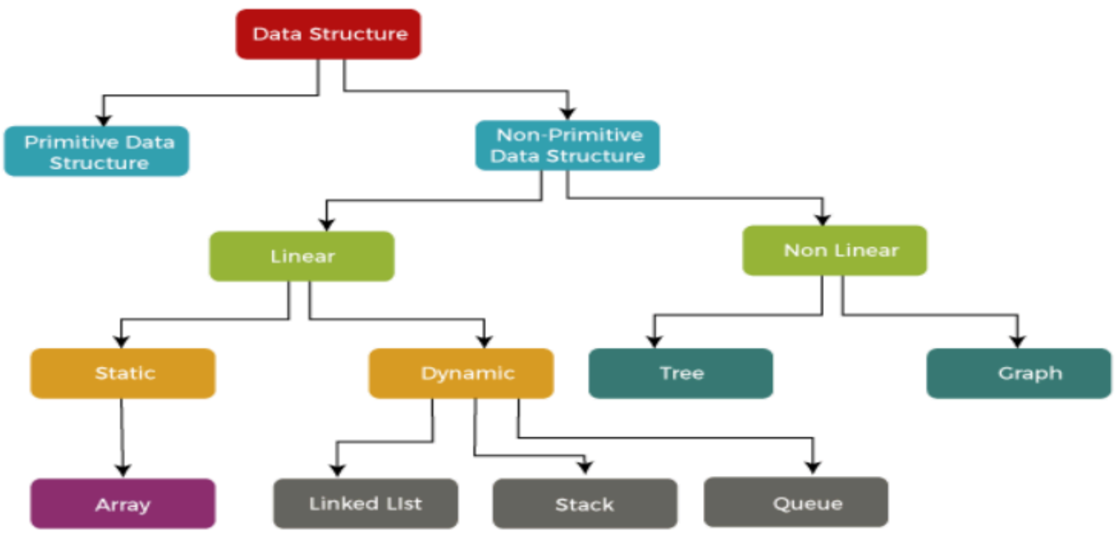
1 **a) What is data structure? Explain the classifications of Data structures with examples.**

Solution:
A **data structure** is a way of organizing, managing, and storing data in a computer so it can be used efficiently. Data structures allow data to be arranged in a way that enables easy access, modification, and processing.

Classification of Data Structures

Data structures can be broadly classified into two main types:

1. **Primitive Data Structures**
2. **Non-Primitive Data Structures**



1. Primitive Data Structures

Primitive data structures are the basic data types provided by programming languages, such as integers, floats, characters, and booleans. These types hold a single value and are usually built into the language.

2. Non-Primitive Data Structures

Non-primitive data structures are more complex and are used to store multiple values in a single structure. They are divided into two main categories: **linear** and **non-linear** data structures.

A. Linear Data Structures

- **Arrays:** A collection of elements, each identified by an index or key. Elements are stored in contiguous memory locations, and all elements are of the same type.
  - *Example:* `[10, 20, 30, 40]`
- **Linked Lists:** A sequence of elements called nodes, where each node contains a value and a reference to the next node. Unlike arrays, elements are not stored in contiguous memory locations.
  - *Example:* `10 -> 20 -> 30 -> 40`
- **Stacks:** A collection of elements that follows the Last-In-First-Out (LIFO) principle. Operations are performed at only one end of the structure (top of the stack).
  - *Example:* A stack of plates where only the top plate is accessible.
- **Queues:** A collection of elements that follows the First-In-First-Out (FIFO) principle. Elements are added at one end (rear) and removed from the other end (front).
  - *Example:* A line of people waiting to buy tickets, where the person at the front of the line is served first.

B. Non-Linear Data Structures

- **Trees:** A hierarchical data structure consisting of nodes, where each node has a value and references to child nodes. Trees are commonly used for data that has a natural hierarchy, like file directories.
  - *Example:* A binary tree representing a family tree, with each node representing a family member.
- **Graphs:** A collection of nodes (vertices) connected by edges. Graphs are used to represent networks and relationships, such as social networks or web page links.
  - *Example:* A social network graph where each person is a node, and an edge represents a friendship.

b) With examples explain pointer declaration, pointer initialization, and void pointer.

Solution:

Pointer Declaration

A **pointer** is a variable that stores the memory address of another variable. When declaring a pointer, we use an asterisk (*) before the pointer's name to denote it as a pointer type.

Syntax: data_type *pointer_name;

Example: int *p;

Pointer Initialization

After declaring a pointer, we need to assign it the address of a variable of the same data type using the address-of operator `&`.

Example: int x = 10; int *p = &x;

Void Pointer:

A void pointer (also known as a generic pointer) is a special type of pointer that can point to any data type. It is declared using the `void` keyword.

Example:

void *vp;

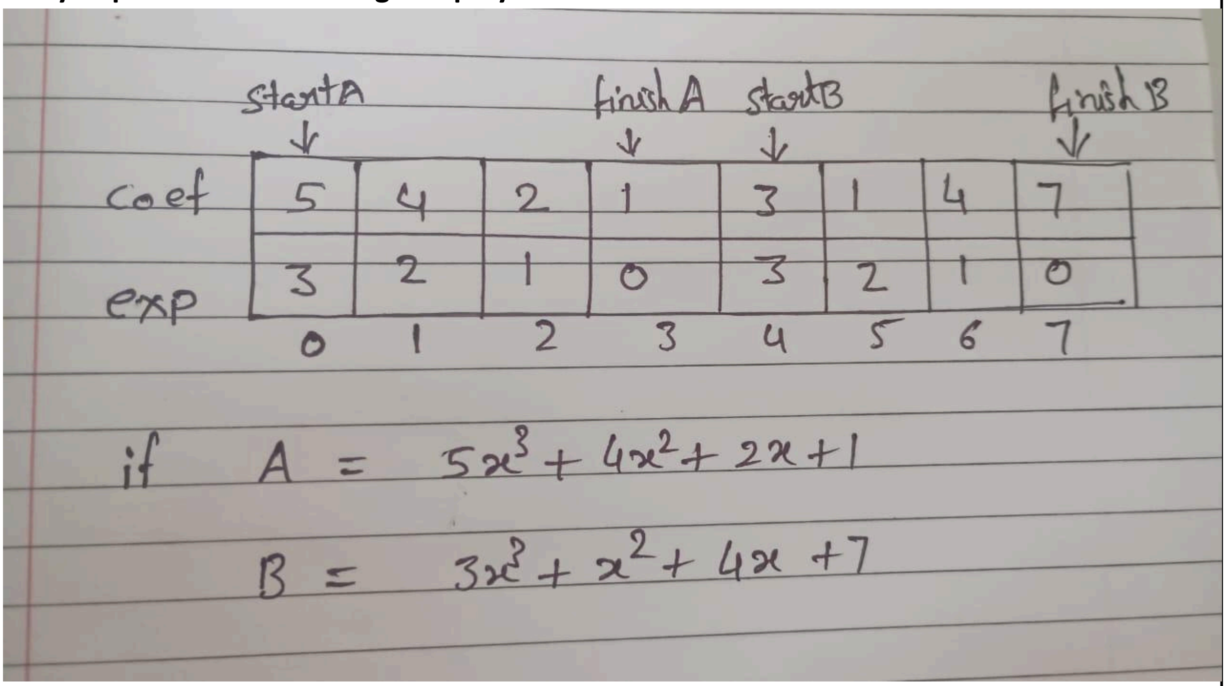
int x = 5;

vp = &x;

2 a) **Represent below polynomial using Array and write a C function to perform polynomial addition:1. $5x^3 + 4x^2 + 2x + 1$ 2. $3x^3 + x^2 + 4x + 7$**

Solution:

Array Representation of the given polynomials:



Polynomial Addition Function in C:

```
int addExpressions(int firstCount, int secondCount)
{
    int i, j, k;
    i = 0;
    j = 0;
    k = 0;
```

```
    while(i < firstCount && j < secondCount)
    {
        if(first[i].exp == second[j].exp)
        {
            result[k].coeff = first[i].coeff + second[j].coeff;
            result[k].exp = first[i].exp;
            i++;
            j++;
            k++;
        }
        else if(first[i].exp > second[j].exp)
        {
            result[k].coeff = first[i].coeff;
            result[k].exp = first[i].exp;
            i++;
            k++;
        }
        else
        {
            result[k].coeff = second[i].coeff;
            result[k].exp = second[j].exp;
            j++;
            k++;
        }
    }

    while(i < firstCount)
    {
        result[k].coeff = first[i].coeff;
        result[k].exp = first[i].exp;
        k++;
        i++;
    }

    while(j < secondCount)
    {
        result[k].coeff = second[j].coeff;
        result[k].exp = second[j].exp;
        k++;
        j++;
    }
    return k;
}
```

b) Evaluate the following postfix expression step by step using stack, based on values given below for each variable: A B + C D - * E +, Where A= 5, B = 2, C= 4, D = 3, E = 6.

Solution:

To evaluate the postfix expression AB+CD−*E+ step-by-step using a stack, let's first substitute the values of the variables:

- A = 5
- B = 2
- C = 4
- D = 3
- E = 6

So, the expression becomes: 5 2 + 4 3 − * 6 +

Step-by-Step Evaluation

1. Start with an empty stack.
2. Read each element from left to right and perform the following:
   - If the element is a number, push it onto the stack.
   - If the element is an operator, pop the required number of operands from the stack, perform the operation, and push the result back onto the stack.

Evaluation Steps:

steps	Expression	Action	Stack
1	5	Push 5 onto the stack	[5]
2	2	Push 2 onto the stack	[5, 2]
3	+	Pop 5 and 2, calculate 5 + 2 = 7, push 7	[7]
4	4	Push 4 onto the stack	[7, 4]
5	3	Push 3 onto the stack	[7, 4, 3]
6	-	Pop 4 and 3, calculate 4 - 3 = 1, push 1	[7, 1]
7	*	Pop 7 and 1, calculate 7 * 1 = 7, push 7	[7]

8	6	Push 6 onto the stack	[7, 6]
9	+	Pop 7 and 6, calculate 7 + 6 = 13, push 13	[13]

Final Result:

After evaluating the entire expression, the final result is the only value left in the stack, which is:13

3 a) Explain Knuth Morris Pattern Matching Algorithm with example.

Knuth Morris Pattern Matching Algorithm:

* It is a string-searching pattern matching algorithm by avoiding redundant comparisons in linear time complexity $O(n+m)$

n - length of string
m - length of Pattern.

* By knowing the characters in the pattern and the position in the pattern where a mismatch occurs with a character in S we can determine where in the pattern to continue the search for a match without moving backward in S.

To achieve this, we use a failure function.

Failure function:

If $p = p_0 p_1 \ldots p_{n-1}$ is a pattern.

failure function is defined as,

$$f(j) = \begin{cases} \text{largest } i < j \text{ such that } p_0 p_1 \cdots p_i = p_{j-i} p_{j-i+2} \cdots p_j & \text{if such an } i \geq 0 \text{ exists} \\ -1 & \text{, otherwise} \end{cases}$$

Example:

Pattern; pat = abcabcacab.

j	0	1	2	3	4	5	6	7	8	9
pat	a	b	c	a	b	c	a	c	a	b
F	-1	-1	-1	0	1	2	3	-1	0	1.

Algorithm:

① if partial metching is found we compare

S_i and $P_{f(j-1)+1}$

② If j=0 we compare

S_{i+1} and P_0.

Failure Function:

```
void fail ()
{
    int n= strlen(pat);
    failure[0] =-1;
    for (j=1; j<n; j++)
    {
        i= failure[j-1];
```

```
while ((pat[j] != pat[i+1]) && (i>=0))
        i = failure[i];

if (pat[j] == pat[i+1]
        failure[j] = i+1;

else
        failure[j] = -1;
}

}
```

KMP - C - function:

```
int pmatch(char *string, char *pat)
{
    int i=0, j=0;
    int lens = strlen(string);
    int lenp = strlen(pat);
    while(i<lens && j<lenp)
    {
        if (string[i] == pat[j])
            i++; j++;
        else if (j==0)
            i++;
        else
            j = failure[j-1]+1;
    }
    return( (j==lenp) ? (i-lenp) : -1);
}
```

b) For the pattern "aabaabaaa" and the text "aabaacaadaabaabaaa", apply the KMP (Knuth-Morris-Pratt) algorithm to search for the pattern in the text.

Failure Function.

0	1	2	3	4	5	6	7	8
a	a	b	a	a	b	a	a	a
-1	0	-1	0	1	2	3	4	0

KMP

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
a	a	b	a	a	c	a	a	d	a	a	b	a	a	b	a	a	a

a a b aa b aaa.

a a b aa b a a a

a a b a a b a a a

Index: 9

4 a) Define stack. Give the C implementation of push and pop functions. Include a check for empty and full conditions of a stack.

Implementation of a stack:

* One-dimensional array is used given as

stack [MAX_SIZE].

where, MAX_SIZE - is the maximum number of entries.

* Pointer, top is set to -1 (initially)
* Empty Stack: top = -1.
* Full stack denotes: top >= (MAX_SIZE - 1)

Draw previous diagram also.

Operations of Stack:

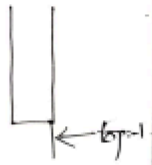
① Push

② Pop.

Status of stack need to be checked before performing operation.

① Underflow

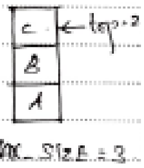
* Underflow represents empty stack.
* top = -1.



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

Overflow:

* Overflow represent full stack
* top >= MAX_SIZE - 1




```
int overflow()
 {
     if (top >= MAX_SIZE - 1)
          return 1;
     else
          return 0;
```

Push Operation:

Inserting data into a stack is called push operation

```
Void push(int ele)
 {
   if (Overflow)
       printf("Stack is full - No Insertion");
   else
       s[++top] = ele;
 }
```

```
int pop()
{
  if (underflow())
      printf("Stack is empty
                No deletion");
  else
      return stack[top--];
```

b). Convert the following infix expression to a postfix expression 1. A+(B*C+D/E)$F

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

- **Step-by-step conversion**:
  - A: Operand → add to output.
  - +: Push onto the stack.
  - (: Push onto the stack.
  - B: Operand → add to output.
  - *: Push onto the stack.
  - C: Operand → add to output.
  - +: Pop * (from the stack) and add to output, then push +.
  - D: Operand → add to output.
  - /: Push onto the stack.
  - E: Operand → add to output.
  -): Pop / and add to output, then pop + and add to output, then discard (.
  - ^: Push onto the stack.
  - F: Operand → add to output.

Postfix for A + (B * C + D / E) ^ F: A B C * D E / + F ^ +

2. P*(Q-R)/X * (S + T)^U^V

   Step-by-step conversion:

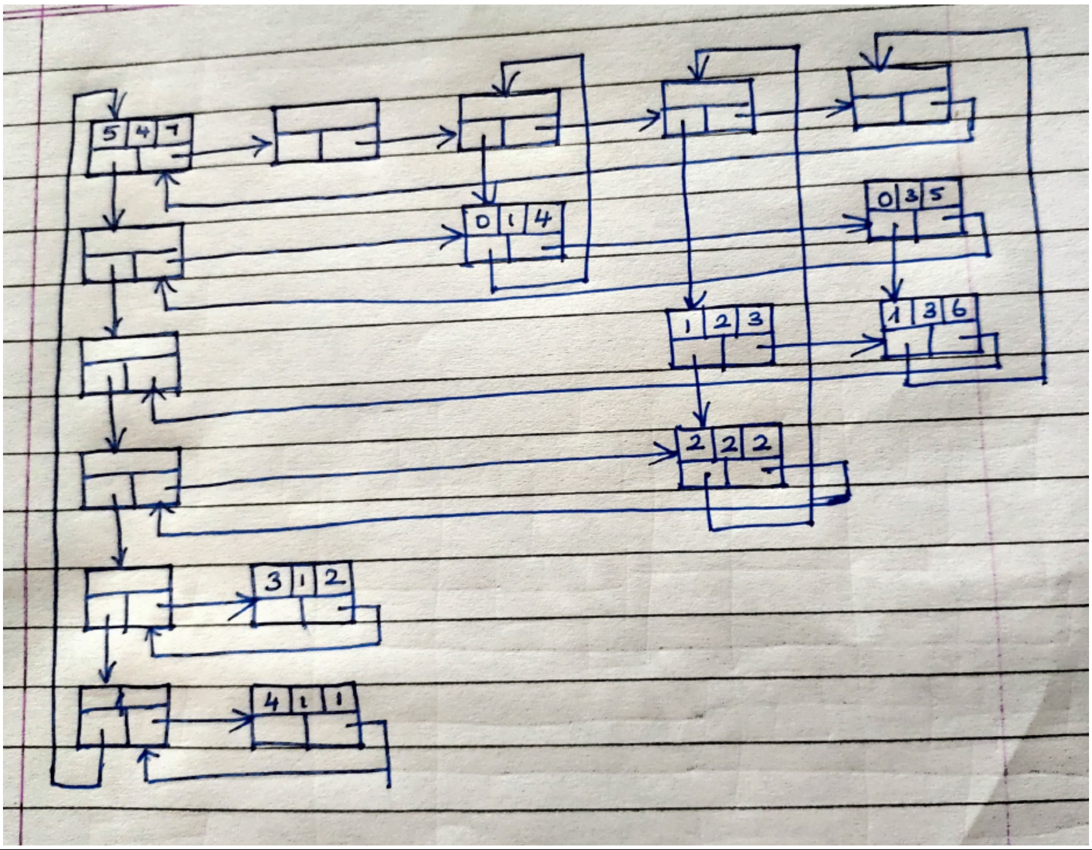
   a. P: Operand → add to output.
   b. *: Push onto the stack.
   c. (: Push onto the stack.
   d. Q: Operand → add to output.
   e. -: Push onto the stack.
   f. R: Operand → add to output.
   g.): Pop - and add to output, then discard (.
   h. /: Push onto the stack.
   i. X: Operand → add to output.
   j. *: Push onto the stack.
   k. (: Push onto the stack.
   l. S: Operand → add to output.
   m. +: Push onto the stack.
   n. T: Operand → add to output.
   o.): Pop + and add to output, then discard (.
   p. ^: Push onto the stack.

q. U: Operand → add to output.
r. ^: Push onto the stack (right-to-left associativity).
s. V: Operand → add to output.

Postfix for **P * (Q - R) / X * (S + T) ^ U ^ V**: P Q R - * X / S T + U V ^ ^ *

5 a) Represent the following matrix in the linked representation form:

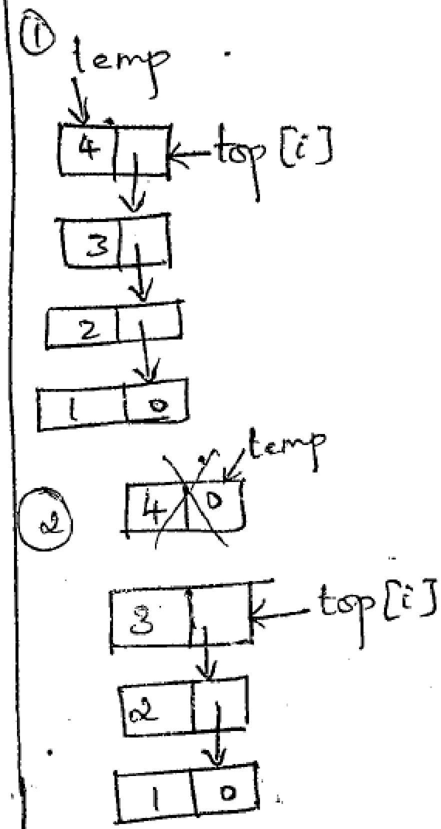
0	4	0	5
0	0	3	6
0	0	2	0
2	0	0	0
1	0	0	0



b) Write a program in C to implement push and pop operation on a stack of integers using a singly linked list.
push:

```
void push(int i, element item)
{
    temp = (stackptr) malloc( sizeof (stackptr));
    temp->data = item;
    temp->link = top[i];
    top[i] = temp;
}
```

```
element pop(int i)
{
    stackptr temp = top[i];
    element item;
    if(!temp)
        return stackEmpty();
    item = temp->data;
    top[i] = temp->link;
    free(temp);
    return item;
}
```



6

a) Write C- functions for the following on a Singly linked list of Char data:

1. Insert a node in the beginning of a list

```
void insert_front(int ele)
{
   create_node(ele);
   if(first)
      temp->next=first;
   first=temp;
}
```

2. delete a node after a node in a list

```
void delete_node(int ele)
{
   ptr=first;
   while(ptr->next->data!=ele)
   {
      ptr=ptr->next;
   }
   temp=ptr->next;
   ptr->next=temp->next;
   free(temp);
}
```

3. display the list.

3. Print a List:

```
void print_list()
{
   for(ptr=first;ptr;ptr=ptr->next)
      printf("%d ->",ptr->data);
   printf("NULL");
}
```

4. To reverse the direction of the singly linked list (as shown below)

start

A	B	C	D

start

D	C	B	A

Reverse a List:

```
void reverse()
{
   temp=prev=NULL;
   ptr=first;
   while(ptr!=NULL)
   {
      prev=ptr;
      ptr=ptr->next;
      prev->next=temp;
      temp=prev;
   }
   sec=temp;
}
```

b) Write a C function to insert a node at 2nd position in DLL.

```
void insertAtSecondPosition(struct Node** head, int data) {
    struct Node* newNode = createNode(data);

    // If the list is empty or has only one node
    if (*head == NULL || (*head)->next == NULL) {
        printf("List has less than 2 nodes. Inserting at the start.\n");
        newNode->next = *head;
        if (*head != NULL) {
            (*head)->prev = newNode;
        }
        *head = newNode;
        return;
```

```
    }

    // Insert the new node at the 2nd position
    struct Node* first = *head;
    struct Node* second = first->next;

    newNode->next = second;
    newNode->prev = first;

    first->next = newNode;
    second->prev = newNode;
}
```