SEM – III Computer Engineering Subject: Data Structures

Choice Based Credit and Grading System (CBCGS)

Note: Please note that this answer key is provided as a reference only. Any alternative solution which gives correct output should be considered equally valid.

N.B: (1) Question No.1 is compulsory
(2) Attempt any three questions of the remaining five questions
(3 Figures to the right indicate full marks
(4) Make suitable assumptions wherever necessary with proper justifications

Q.1 (a) Explain ADT. List the Linear and Non-linear data structures with example (5)

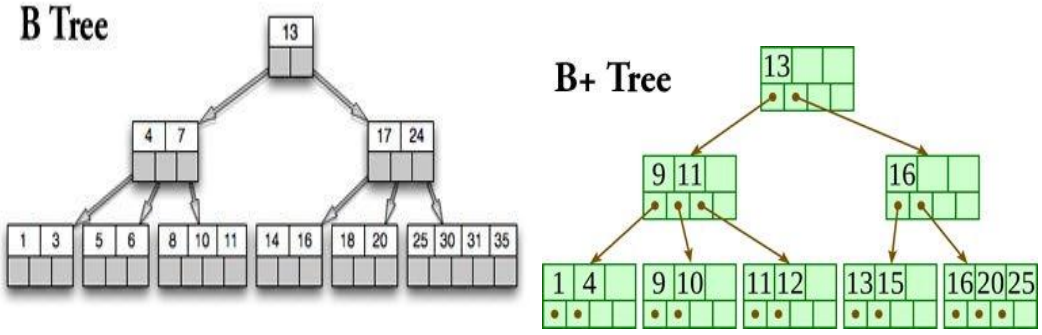
***ADT** refers to an **A**bstract **D**ata **T**ype. This focuses on the behavior of a data structure rather than on any implementation details.*

***Linear data structures**- the data elements are organized in some sequence is called linear data structure. Here the various operations on a data structure are possible only in a sequence i.e. we cannot insert the element into any location of our choice. Examples of linear data structures are array, stacks, queue, and linked list.*

***Non-Linear data structures** -When the data elements are organized in some arbitrary function without any sequence, such data structures are called non-linear data structures. Examples of such type are trees, graphs.*

(b) Explain B Tree and B+ Tree. (5)

***A B-tree** is a generalized binary search tree, in which any node can have more than two children. Each internal node in a B-tree contains a number of keys. These keys separate the values, and further forms the sub-trees.*



***A B+ tree** is an n-array tree with a node, which consists of a large number of children per node. The root may be a leaf or a node that contains more than two children. A B+ tree consists of a root, internal nodes and leaves.*

(c) Write a program to implement Binary Search on sorted set of Integers (10)

```
#include <stdio.h>
#include <conio.h>
main()
{
   intarr[10], num, i, n , pos =-1, beg, end,mid, found =0;
```

```
  clrscr ();
  printf("\n Enter the number of elements in the array: ");
  scanf ("%d", &n);
  printf (" \n Enter the elements: ");
 for(i=0;i<n;i++)
 {
   scanf("%d", &arr[i]);
}
   printf("\n Enter the number that has to be searched: " );
   scanf ("%d", &num);
   beg = 0, end = n-1;
  while (beg <end)
  {
    mid= (beg+ end)/2;
     if (arr[mid] == num)
     {
       printf("\n %dis present in the array at position = %d", num, mid);
       found=1;
       break;
     }
        if (arr[mid]>num)
        {
          end = mid-1;
        }
       else if (arr[mid] <num)
      beg = mid+1;
  }
    if ( beg > end &&found == 0)
    {
      printf("\n %d DOESNOTEXIST IN THE ARRAY",num);
    }
       getch();
       return 0;
}
```

Q.2(a) Write a program to convert Infix expression into Postfix expression. (10)

```
#include<stdio.h>
char stack[20];
int top = -1;
void push(char x)
{
stack[++top] = x;
}
char pop()
{
if(top == -1)
return -1;
else
return stack[top--];
```

```
}
int priority(char x)
{
if(x == '(')
return 0;
if(x == '+' || x == '-')
return 1;
if(x == '*' || x == '/')
return 2;
}

main()
{
charexp[20];
char *e, x;
printf("Enter the expression :: ");
scanf("%s",exp);
   e = exp;
while(*e != '\0')
   {
if(isalnum(*e))
printf("%c",*e);
else if(*e == '(')
push(*e);
else if(*e == ')')
      {
while((x = pop()) != '(')
printf("%c", x);
      }
else
      {
while(priority(stack[top]) >= priority(*e))
printf("%c",pop());
push(*e);
      }
e++;
   }
while(top != -1)
   {
printf("%c",pop());
   }
}
```

(b)Explain Huffman Encoding with an example (10)

Huffman's algorithm is based on the idea that a variable length code should use the shortestcode words for the most likely symbols and the longest code words for the least likely symbols. Inthis way, theaverage Code length will be reduced. The algorithm assigns code words to symbols byconstructing a binary coding tree. Each symbol of the alphabet is a leaf of the coding tree. Thecode of a given symbol corresponds to the unique path from the root to that leaf, with 0 or 1 addedto the code for each edge along the path depending on whether the left or right child of a givennode occurs next along the path.

Let $\mathcal{A} = \{a_1, \ldots, a_5\}$, $P(a_i) = \{0.2, 0.4, 0.2, 0.1, 0.1\}$.

Symbol	Step 1	Step 2	Step 3	Step 4	Codeword
a_2	0.4	0.4	0.4	0.6 0	1
a_1	0.2	0.2	0.4 0	0.4 1	01
a_3	0.2	0.2 0	0.2 1		000
a_4	0.1 0	0.2 1			0010
a_5	0.1 1				0011

Combine last two symbols with lowest probabilities, and
use one bit (last bit in codeword) to differentiate between them!

Q.3 (a) Write a program to implement Doubly Linked List. Provide the following operations: (10)

(i) Insert a node in the beginning

(ii) Insert a node in the end.

(iii) Delete a node from the end

(iv) Display the list

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
struct node *previous;
int data;
struct node *next;
}*head, *last;

voidinsert_begning(int value)
{
struct node *var,*temp;
var=(struct node *)malloc(sizeof(struct node));
var->data=value;
if(head==NULL)
    {
head=var;
head->previous=NULL;
head->next=NULL;
last=head;
    }
else
    {
temp=var;
temp->previous=NULL;
temp->next=head;
head->previous=temp;
```

```
head=temp;
    }
}

voidinsert_end(int value)
{
struct node *var,*temp;
var=(struct node *)malloc(sizeof(struct node));
var->data=value;
if(head==NULL)
    {
head=var;
head->previous=NULL;
head->next=NULL;
last=head;
    }
else
    {
last=head;
while(last!=NULL)
        {
temp=last;
last=last->next;
        }
last=var;
temp->next=last;
last->previous=temp;
last->next=NULL;
    }
}

intdelete_from_end()
{
struct node *temp;
temp=last;
if(temp->previous==NULL)
     {
free(temp);
head=NULL;
last=NULL;
return 0;
     }
printf("\nData deleted from list is %d \n",last->data);
last=temp->previous;
last->next=NULL;
free(temp);
return 0;
}


void display()
```

```
{
struct node *temp;
temp=head;
if(temp==NULL)
    {
printf("List is Empty");
    }
while(temp!=NULL)
   {
printf("-> %d ",temp->data);
temp=temp->next;
   }
}

int main()
{
int value, i;
head=NULL;
printf("Select the choice of operation on link list");
printf("\n1.) insert at begning\n2.) insert at at end\");
printf("\n3.) delete from end\n4.) display list\n5.)exit");
while(1)
  {
printf("\n\nenter the choice of operation you want to do ");
scanf("%d",&i);
switch(i)
       {
case 1:
           {
printf("enter the value you want to insert in node ");
scanf("%d",&value);
insert_begning(value);
display();
break;
           }
case 2:
           {
printf("enter the value you want to insert in node at last ");
scanf("%d",&value);
insert_end(value);
display();
break;
           }
case 3:
           {
delete_from_end();
display();
break;
           }
case 4:
           {
```

```
display();
break;
            }
case 5 :
            {
exit(0);
break;
            }
        }
    }
printf("\n\n%d",last->data);
display();
getch();
}
```

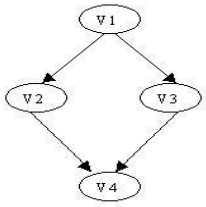
(b) Explain Topological sorting with example (10)

Topological sort*: an ordering of the vertices in a directed acyclic graph, such that:*
*If there is a path from **u** to **v**, then **v** appears after **u** in the ordering.*

Types of graphs:

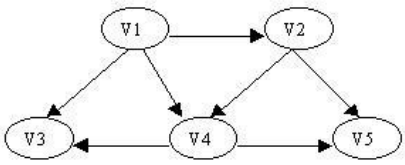
*The graphs should be **directed:** otherwise for any edge (u,v)*
*there would be a path from **u** to **v** and also from **v** to **u**,*
and hence they cannot be ordered.
*The graphs should be **acyclic**: otherwise for any two vertices **u** and **v** on a cycle*
u** would precede **v** and **v** would precede **u.

The ordering may not be unique:



V1, V2, V3, V4 and V1, V3, V2, V4 are legal orderings
Algorithm

1. *Compute the indegrees of all vertices*
2. *Find a vertex **U** with indegree 0 and print it (store it in the ordering)*
   If there is no such vertex then there is a cycle
   and the vertices cannot be ordered. Stop.
3. *Remove **U** and all its edges **(U,V)** from the graph.*
4. *Update the indegrees of the remaining vertices.*
5. *Repeat steps 2 through 4 while there are vertices to be processed.*



	Indegree					
Sorted à		*V1*	*V1,V2*	*V1,V2,V4*	*V1,V2,V4,V3*	*V1,V2,V4,V3,V5*
V1	*0*					
V2	*1*	*0*				
V3	*2*	*1*	*1*	*0*		
V4	*2*	*1*	*0*			
V5	*2*	*2*	*1*	*0*	*0*	

One possible sorting: V1, V2, V4, V3, V5

Another sorting: V1, V2, V4, V5, V3

Q.4 (a) Write a program to implement Quick sort. Show the steps to sort the given numbers: (10)

25, 13, 7, 34, 56,23,13,96,14,2

```
#include <stdio.h>

voidquick_sort(int[],int,int);
int partition(int[],int,int);

int main()
{
   int a[50],n,i;
   printf("How many elements?");
   scanf("%d",&n);
   printf("\nEnter array elements:");

   for(i=0;i<n;i++)
      scanf("%d",&a[i]);

   quick_sort(a,0,n-1);
   printf("\nArray after sorting:");

   for(i=0;i<n;i++)
      printf("%d ",a[i]);

   return 0;
}

voidquick_sort(int a[],intl,int u)
{
   int j;
   if(l<u)
   {
      j=partition(a,l,u);
```

```
        quick_sort(a,l,j-1);
        quick_sort(a,j+1,u);
    }
}

int partition(int a[],intl,int u)
{
    intv,i,j,temp;
    v=a[l];
    i=l;
    j=u+1;

    do
    {
        do
            i++;

        while(a[i]<v&&i<=u);

        do
            j--;
        while(v<a[j]);

        if(i<j)
        {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
        }
    }while(i<j);

    a[l]=a[j];
    a[j]=v;

    return(j);
}
```

(b) Write a program to implement linear queue using arrays (10)

```
#include <stdio.h>
#include <malloc.h>
#define MAX 10
int queue[MAX];
int front = -1, rear = -1;
void insert(void);
intdelete_element(void);
int peek(void);
void display(void);
int main()
{
```

```
int choice, val;
do {
printf("\n1. Insert an element into a queue ");
printf("\n2. Delete an element from a queue ");
printf("\n3. Peek an element form a queue ");
printf("\n4. Display the queue ");
printf("\n5. EXIT");
printf("\n\n\nEnter your choice : ");
scanf("%d",&choice);
switch(choice)
{
case 1:
insert();
break;
case 2:
val = delete_element();
if(val != -1);
printf("\n The number deleted is : %d", val);
break;
case 3:
val = peek();
if(val != -1);
printf("\n The first value in queue is : %d", val);
break;
case 4:
display();
break;
}
} while(choice != 5);
return 0;
}
void insert() {
intnum;
printf("\n Enter the element to be inserted into the queue : ");
scanf("%d", &num);
if(rear == MAX-1)
printf("\n OVERFLOW");
else if(front == -1 && rear == -1)
front = rear = 0;
else
rear++;
queue[rear] = num;
}
intdelete_element()
{
intval;
if(front == -1 || front > rear)
{
printf("\n UNDERFLOW");
return -1;
}
```

```
else {
val = queue[front];
front++;
if(front > rear)
front = rear = -1;
returnval;
}
}
int peek()
{
if(front == -1 || front > rear)
{
printf("\n QUEUE IS EMPTY");
return -1;
}
else {
return queue[front];
}
}
void display()
{
inti;
printf("\n");
if(front == -1 || front > rear )
printf("\n QUEUE IS EMPTY");
else {
for(i = front;i<= rear ; i++)
printf("\t %d", queue[i]);
}
}
```

Q.5. (a) Write a program to implement STACK using Linked List. What are the advantages of linked list overarray? (10)

```
#include <stdio.h>
#include <stdlib.h>

struct Node
{
int Data;
struct Node *next;
}*top;

voidpopStack()
{
struct Node *temp, *var=top;
if(var==top)
   {
top = top->next;
free(var);
```

```
   }
else
printf("\nStack Empty");
}

void push(int value)
{
struct Node *temp;
temp=(struct Node *)malloc(sizeof(struct Node));
temp->Data=value;
if (top == NULL)
   {
top=temp;
top->next=NULL;
   }
else
   {
temp->next=top;
top=temp;
   }
}

void display()
{
struct Node *var=top;
if(var!=NULL)
    {
printf("\nElements are as:\n");
while(var!=NULL)
        {
printf("\t%d\n",var->Data);
var=var->next;
        }
printf("\n");
    }
else
printf("\nStack is Empty");
}

int main(intargc, char *argv[])
{
inti=0;
top=NULL;
printf(" \n1. Push to stack");
printf(" \n2. Pop from Stack");
printf(" \n3. Display data of Stack");
printf(" \n4. Exit\n");
while(1)
    {
printf(" \nChoose Option: ");
scanf("%d",&i);
```

```
switch(i)
        {
case 1:
            {
int value;
printf("\nEnter a valueber to push into Stack: ");
scanf("%d",&value);
push(value);
display();
break;
            }
case 2:
            {
popStack();
display();
break;
            }
case 3:
            {
display();
break;
            }
case 4:
            {
struct Node *temp;
while(top!=NULL)
            {
temp = top->next;
free(top);
top=temp;
            }
exit(0);
            }
default:
            {
printf("\nwrong choice for operation");
            }
        }
    }
}
```

Dynamic size: As the size of linked list is not fixed so we can add or remove as much elements as required. But in array we have to pre-define the array size which we can't change later.

Ease of insertion/deletion: Inserting a new element in an array of elements is expensive; because room has to be created for the new elements and to create room existing elements have to shift.

(b) Write a program to implement Binary Search Tree (BST). Show BST for the following input: (10)

10, 5, 4, 12, 15, 11, 3

```
#include <stdio.h>
#include <stdlib.h>

structTreeNode {
int data;
structTreeNode *leftChildNode;
structTreeNode *rightChildNode;
};

typedefstructTreeNode node;
node *rootNode = NULL;

voidinsertNode(inti, node *n) {
if(n == NULL) {
      n = (node*)malloc(sizeof(node));
n->leftChildNode = NULL;
n->rightChildNode = NULL;
n->data = i;
   }
else
if(n->data == i)
printf("\nThis value already exists in the tree!");
else
if(i> n->data)
insertNode(i, n->rightChildNode);
else
insertNode(i, n->leftChildNode);
   }

voidsearchNode(inti, node *n) {
if(n == NULL)
printf("\nValue does not exist in tree!");
else
if(n->data == i)
printf("\nValue found!");
else
if(i> n->data)
searchNode(i, n->rightChildNode);
else
searchNode(i, n->leftChildNode);
   }



voiddisplayPreOrder(node *n) {
if(n != NULL) {
printf("%d ", n->data);
displayPreOrder(n->leftChildNode);
displayPreOrder(n->rightChildNode);
```

```
   }
}

voiddisplayPostOrder(node *n) {
if(n != NULL) {
displayPostOrder(n->leftChildNode);
displayPostOrder(n->rightChildNode);
printf("%d ", n->data);
   }
}

voiddisplayInOrder(node *n) {
if(n != NULL) {
displayInOrder(n->leftChildNode);
printf("%d ", n->data);
displayInOrder(n->rightChildNode);
   }
}

int main(void) {
intch, num, num1;
do {
printf("\nSelect a choice from the menu below.");
printf("\n1. Insert a node.");
printf("\n2. Search for a node.");
printf("\n3. Display the Binary Search Tree.");
printf("\nChoice: ");
scanf("%d", &ch);
switch(ch) {
case 1: printf("\nEnter an element: ");
scanf("%d", &num);
                //printf("YESYES");
insertNode(num, rootNode);
break;

case 2: printf("\nEnter the element to be searched for: ");
scanf("%d", &num);
searchNode(num, rootNode);
break;

case3: printf("\nSelect an order for the elements to be display in.");
printf("\n1. Pre-order.");
printf("\n2. Post-order.");
printf("\n3. In-order.");
printf("\nChoice: ");
scanf("%d", &num1);
switch(num1) {
case 1: printf("\nPre-order Display: ");
displayPreOrder(rootNode);
break;
```

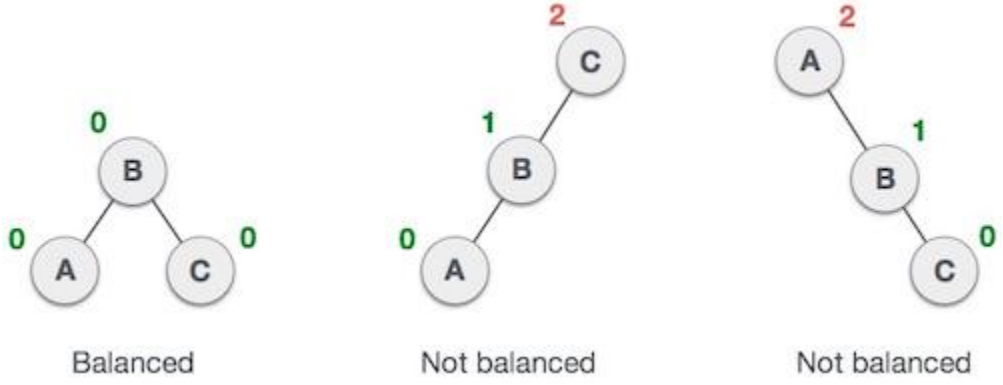
```
case 2: printf("\nPost-order Display: ");
displayPostOrder(rootNode);
break;

case 3: printf("\nIn-order Display: ");
displayInOrder(rootNode);
break;

default: exit(0);
                }
break;

default: exit(0);
        }
      //printf("%d", rootNode->data);
printf("\nIf you want to return to the menu, press 1.");
printf("\nChoice: ");
scanf("%d", &num);
   } while(num == 1);
return 0;
```

Q.6. Write Short notes on (any two) (20)

(a) AVL Tree

*Named after their inventor **Adelson**, **Velski**&**Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.*

Here we see that the first tree is balanced and the next two trees are not balanced −



*In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.*

***BalanceFactor** = height(left-sutree) − height(right-sutree)*

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

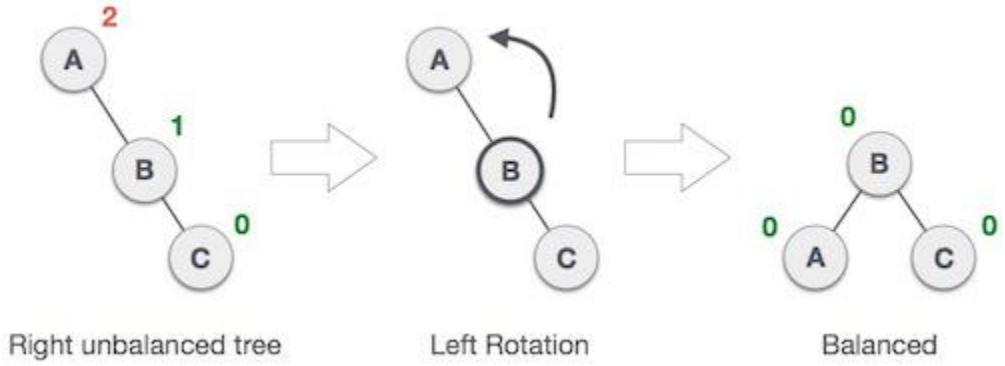
To balance itself, an AVL tree may perform the following four kinds of rotations −

- *Left rotation*
- *Right rotation*
- *Left-Right rotation*
- *Right-Left rotation*

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

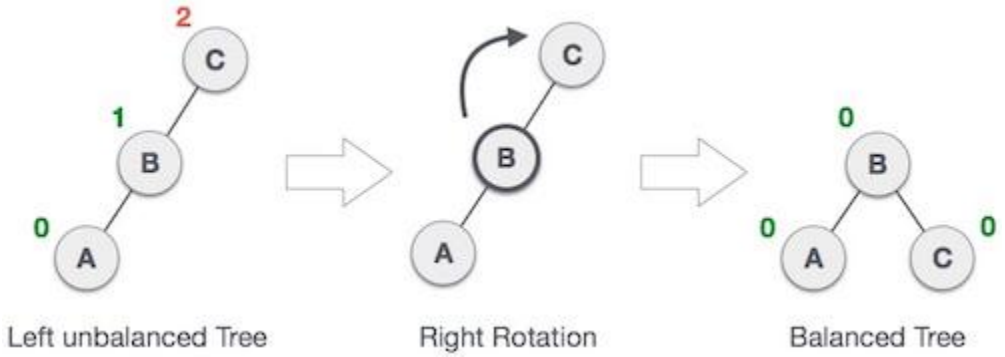
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



*In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.*

Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

(b) Graph Traversal Techniques

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process. A graph traversal finds the egdes to be used in

the search process without creating loops that means using graph traversal we visit all vertices of graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. ***DFS (Depth First Search)***
2. ***BFS (Breadth First Search)***

DFS (Depth First Search)

*DFS traversal of a graph, produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.*

We use the following steps to implement DFS traversal...

- ***Step 1:** Define a Stack of size total number of vertices in the graph.*
- ***Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.*
- ***Step 3:** Visit any one of the **adjacent** vertex of the verex which is at top of the stack which is not visited and push it on to the stack.*
- ***Step 4:** Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.*
- ***Step 5:** When there is no new vertex to be visit then use **back tracking** and pop one vertex from the stack.*
- ***Step 6:** Repeat steps 3, 4 and 5 until stack becomes Empty.*
- ***Step 7:** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph*

BFS (Breadth First Search)

*BFS traversal of a graph, produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.*

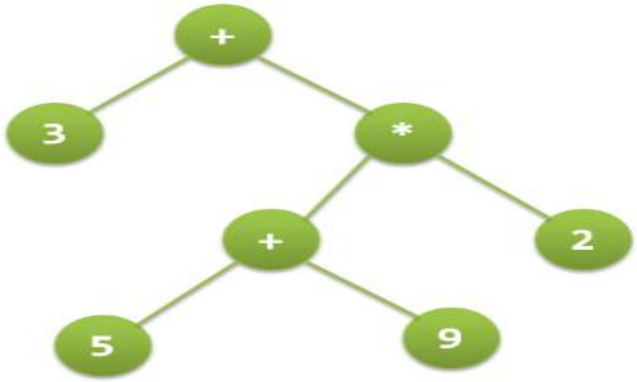
We use the following steps to implement BFS traversal...

- ***Step 1:** Define a Queue of size total number of vertices in the graph.*
- ***Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.*
- ***Step 3:** Visit all the **adjacent** vertices of the verex which is at front of the Queue which is not visited and insert them into the Queue.*
- ***Step 4:** When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.*
- ***Step 5:** Repeat step 3 and 4 until queue becomes empty.*
- ***Step 6:** When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph*

(c) Expression Trees

Expression Tree

*Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for 3 + ((5+9)*2) would be:*



Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression)

Construction of Expression Tree:

Now For constructing expression tree we use a stack. We loop through input expression and do following for every character.

1) If character is operand push that into stack

2) If character is operator pop two values from stack make them its child and push current node again.

At the end only element of stack will be root of expression tree.

(d) Application of Linked list- Polynomial Addition.

Representation of a Polynomial: *A polynomial is an expression that contains more than two terms. A term is made up of coefficient and exponent. An example of polynomial is*

$P(x) = 4x^3+6x^2+7x+9$

A polynomial thus may be represented using arrays or linked lists. Array representation assumes that the exponents of the given expression are arranged from 0 to the highest value (degree), which is represented by the subscript of the array beginning with 0. The coefficients of the respective exponent are placed at an appropriate index in the array. The array representation for the above polynomial expression is given below:

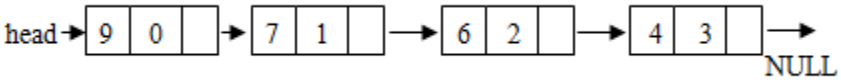
arr	9	7	6	4	(coefficients)
	0	1	2	3	(exponents)

A polynomial may also be represented using a linked list. A structure may be defined such that it contains two parts- one is the coefficient and second is the corresponding exponent. The structure definition may be given as shown below:

```
    struct polynomial
{
int coefficient;
int exponent;
```

*struct polynomial *next;*
};

Thus the above polynomial may be represented using linked list as shown below:



Addition of two Polynomials:

For adding two polynomials using arrays is straightforward method, since both the arrays may be added up element wise beginning from 0 to n-1, resulting in addition of two polynomials. Addition of two polynomials using linked list requires comparing the exponents, and wherever the exponents are found to be same, the coefficients are added up. For terms with different exponents, the complete term is simply added to the result thereby making it a part of addition result.