21CS32-Data Structures and Applications Answer Key-Jan/Feb 2023

MODULE 1

1. a. what is linear array? Discuss the representation of linear array in memory.

Ans:

A linear array, is a list of finite numbers of elements stored in the memory. In a linear array, we can store only homogeneous data elements. Elements of the array form a sequence or linear list, that can have the same type of data.

Representation of Linear Arrays in Memory

The elements of linear array are stored in consecutive memory locations. The computer does not keep track of address of each element of array. It only keeps track of the base address of the array and based on this base address the address or location of any element can be found. We can find out the location of any element by using following formula:

LOC (LA [K]) = Base (LA) + w (K – LB)

Here,

LA is the linear array.

LOC (LA [K]) is the location of the Kth element of LA.

Base (LA) is the base address of LA. w is the number of bytes taken by one element. K is the Kth element. LB is the lower bound.

Suppose we want to find out LOC (A [3]). Here, Base (A) = 1000 w = 2 bytes (Because an integer takes two bytes in the memory). K = 3 LB=0 After putting these values in the given formula, we get: LOC (A [3]) = 1000 + 2 (3 – 0) = 1000 + 2 (3) = 1000 + 6 = 1006.

b. Differentiate between static and dynamic allocations. Discuss four dynamic memory allocation functions.

Ans:

Difference between static and dynamic allocations:

S. No	Static Memory Allocation	Dynamic Memory Allocation
1	In the static memory allocation, variables get allocated permanently, till the program executes or function call finishes.	In the Dynamic memory allocation, variables get allocated only if your program unit gets active.
2	Static Memory Allocation is done before program execution.	Dynamic Memory Allocation is done during program execution.
3	It uses stack for managing the static allocation of memory	It uses heap for managing the dynamic allocation of memory

Memory Allocation (malloc)

• When a malloc function is invoked requesting for memory, it allocates a block of memory that contains the number of bytes specified in its parameter and returns a pointer to the start of the allocated memory.

• When the requested memory is not available the pointer NULL is returned.

syntax: void *malloc (size_t size);

Example: pint=(int*) malloc(sizeof(int));

Contiguous memory allocation (calloc)

• This function is used to allocate contiguous block of memory. It is primarily used to allocate memory for arrays.

• The function calloc () allocates a user specified amount of memory and initializes the allocated memory to 0.

• A pointer to the start of the allocated memory is returned.

• In case there is insufficient memory it returns NULL

syntax: void * calloc (size_t count, size_t size);

Example: int *ptr ptr=(int*) calloc (n, sizeof(int))

Reallocation of memory(realloc)

The function realloc resizes the memory previously allocated by either malloc or calloc.

syntax: Void * realloc (void * ptr, size_t new size);

Example int *p; p=(int*) calloc (n, sizeof(int)) p=realloc (p, s);

Releasing memory (free)

When memory locations allocated are no longer needed, they should be freed by using the predefined function free. Syntax: free(void*);

Example: int *p, a; p=&a; free(p);

c. Write a menu driven program in C for the following array operations:

I) Inserting an element at a given valid position.

2) Deleting an element at a given valid position.

3)Display of array elements.

4)Exit.

Support the program with functions for each of the above operations.

Ans:

```
#include<stdio.h>
int n=0, a [50];

void addLoc () {
   int pos, ele, i;
   printf ("Enter the position\n");
   scanf ("%d", pos);
   pos=pos-1;
```

```
    if(pos>n)
    {
       printf ("\n Location exceeds array size\n");
       return;
    }
    printf ("Enter the element\n");
    scanf ("%d", ele);
    if (pos ==n) // add as the last element of the array
    {
       a[pos]=ele;
       n++;
       return;
    }
    for (i=n-1; i>=pos; i--)
    {
       a[i+1] =a[i]; // shift array elements to the right
    }
    a[pos]=ele;
    n++;
}
void deleteLoc ()
{
    int pos, i;
    printf ("Enter the position\n");
    scanf ("%d", pos);
    pos=pos-1;
```

```
    if(pos>=n)
    {
       printf ("\n Location exceeds array size\n");
       return;
    }
    if(pos==n-1)   //delete last element of the array
       n--;
    else
    {
       for (i=pos; i<=n-1; i++)
       {
          a[i]=a[i+1]; //shift array elements to the left
       }
        n--;
    }
}
void display ()
{
    int i=0;
    if(n==0)
    {
       printf ("array is empty\n");
       return;
    }
    else
    {
```

```
      printf ("array elements are:\n");
      for (i=0; i<n; i++)
          printf ("%d\t", a[i]);
   }
}
void main ()
{
   int i, ch;
   printf ("\n Enter the size of the array\n");
   scanf ("%d", &n);
   printf ("Enter array elements\n");
   for (i=0; i<n; i++)
      scanf ("%d", &a[i]);
   for (; ;) {

printf("\n************************MENU************************");
   printf ("\n1. Inserting an Element at a given valid Position\n2. Deleting an
Element at a given valid Position\n3. Display\n4. Exit\n");

printf("\n*****************************************************\n
");
   printf ("\n Enter your choice\n");
    scanf ("%d", &ch);
    switch(ch)
{
    case 1: addLoc ();
          break;
```

```
    case 2: deleteLoc ();
          break;
    case 3: display ();
          break;
     case 4: exit (0);
    default: printf ("\n Invalid Choice\n");
   }
  }
```

2 a. Give Abstract Data Type for arrays. How array can be declared and initialized?

Ans:

The array is a basic abstract data type that holds an ordered collection of items accessible by an integer index. These items can be anything from primitive types such as integers to more complex types like instances of classes.

Declaring an Array

To declare an array,

1.We should specify the datatype of the array

2.Name of the array

3.And required size with in square brackets.

To declare a single dimensional array,

Example

datatype array name[size];

Where, Datatype can be int, float, double, char etc.

array name should be a valid variable name.

size is an integer value.

Initialization of an Array

We can initialize an array while declaring it. Like below,

Example

int arr [5] = {10,20,30,40,50};

b. With suitable example, discuss self-referential structures.

Ans:

Self-Referential Structures

A self-referential structure is one in which one or more of its data members is a pointer

or

to itself. They require dynamic memory allocation (malloc and free) to explicitly obtain and release memory.

These are the structures in which one or more pointers point to the structure of the

same type.

A Self-Referential Structure means when a structure is referencing another

structure of same type.

Example:

```
struct self {
int p;
struct self *ptr; //It is a pointer to the same structure.
```

} ;

Each instance of the structure self will have two components, p and ptr. p is a single integer variable, while ptr is a pointer to a self-structure.

The value of ptr is either the address in memory of an instance of self or the null pointer.

c. Define sparse matrix. How to represent a sparse matrix? Write an algorithm/function to transpose a given sparse matrix.

Ans:

Sparse Matrix: If a matrix contains more zero entities, then such a matrix is called a sparse matrix.

Ex:

[0 5 0 4 0 0

2 0 0 0 1 0

0 5 0 0 1 0

5 0 0 0 0 0

0 0 0 0 1 0

0 0 0 0 0 1]

The concept of sparse matrix is used in scientific or engineering applications.

Representation:

Array representation: 2D array having n-rows and 3-coloumns, where n is number of non-zero elements.

a[i][0] – represents row value

a[i][1] – represents column value

a[i][2] – represents matrix value

1 2 5

1 4 4

2 1 2

2 5 1

3 2 5

3 5 1

4 5 1

5 6 1

ALGORITHM:

1. Declare and initialize a 2-D array p[a][b] of order axb.
2. Read the matrix p[a][b] from the user.
3. Declare another 2-dimensional array t to store the transpose of the matrix. This array will have the reversed dimensions as of the original matrix.
4. The next step is to loop through the original array and convert its rows to the columns of matrix t.
5. Declare 2 variables i and j.
6. Set both i, j=0
7. Repeat until i<b
8. Repeat until j<a
9. t[i][j] = p[j][i]

j=j+1**

i=i+1

10. The last step is to display the elements of the transposed matrix t.

MODULE 2

3 a. Define stack. Discuss how to implement stack using dynamic arrays.

Stacks Using Dynamic Arrays

If we do not know the maximum size of the stack at compile time, space can be allocated for the elements dynamically at run time and the size of the array can be increases as needed.

Creation of stack: Here the capacity of the stack is taken as 1. The value of the capacity can be altered specific to the application

StackCreateS() ::=
int *stack
Stack=(int*)malloc(stack, sizeof(int)); int capacity = 1;
int top = -1;
BooleanIsEmpty(Stack) ::= top < 0;
BooleanIsFull(Stack) ::= top >= capacity-1;

The function push remains the same except that MAX_STACK_SIZE is replaced with capacity

void **push**(element item)
{
if (top >=capacity-1) stackFull(); stack[++top] = item;
}
The code for the pop function remains unchanged element

pop()
{/* delete and return the top element from the stack */ if (top == -1)
return stackEmpty(); /* returns an error key */ return stack[top--];
}

Stackfull with Array doubling:
The new code for stackFull attempts to increase the capacity of the array stack so that we can add an additional element to the stack. In array doubling, the capacity of the array is doubled whenever it becomes necessary to increase the capacity of an array.

void stackFull()
{
stack=(int*)realloc(stack, 2 * capacity * sizeof(int)) capacity =capacity * 2;
}

b. Write a menu-driven C program for the following operations on stack of integers:

1. Push an element into stack.
2. Pop an element from the stack.
3. Display the contents of stack.

4. Exit.

Show the overflow and underflow conditions.

Ans:

```
#include<stdio.h>
#define MAX 4
int top=-1;
int s[MAX];
void push (int ele)
{

   if (top==MAX-1)
      printf ("\n Stack overflow\n");
   else
      s[++top] =ele;
}

void pop ()
{
   if (top==-1)
      printf ("\n Stack empty\n");
   else
      printf ("\n %d is popped\n", s[top--]);
}

void display ()
{
```

```
    int i;
    if (top==-1)
        printf ("Stack empty\n");
    else
    {
        printf ("Stack elements are: \n");
        for (i=top; i>-1; i--)
        {
            printf ("%d\n", s[i]);
        }
    }
}

void main ()
{
    int ch, x;

    for (; ;)
    {
    printf ("1. Push\n 2. Pop\n 3.Display\n4.Exit\n");
    printf ("Enter your choice\n");
    scanf ("%d", &ch);
    switch(ch)
    {
        case 1: printf ("Enter the value to be pushed to stack\n");
        scanf ("%d", &x);
```

```
        push(x);
        break;

        case 2: pop ();
        break;

        case 3: display ();
        break;

        case 4: exit (0);

        default: printf ("Invalid choice");
    }
   }
}
```

c. What are the disadvantages of ordinary queue? Discuss the implementation of circular queues using arrays.

Ans:

Disadvantage Simple Queue

When the first element is serviced, the front is moved to next element. However, the

position vacated is not available for further use. Thus, we may encounter a situation,

wherein program shows that queue is full, while all the elements have been deleted are

available but unusable, though empty.

Circular queues using arrays:

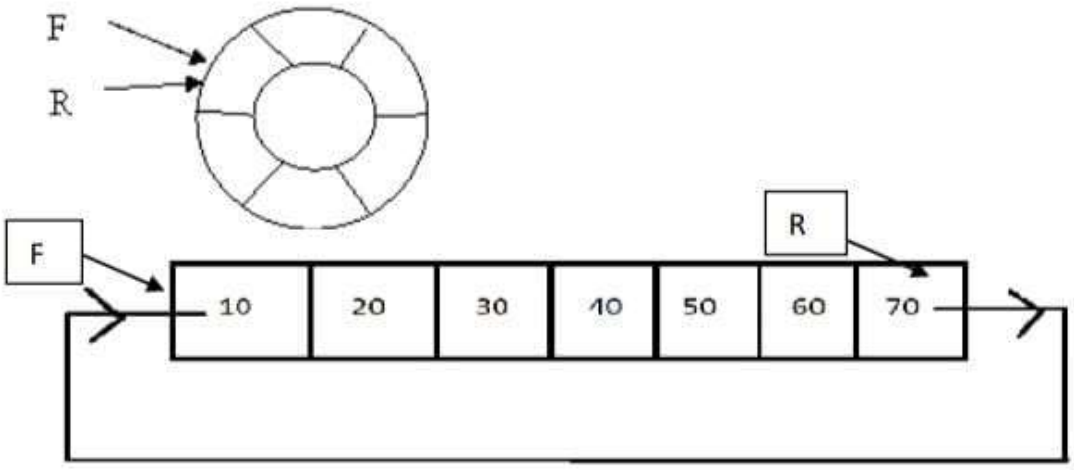
In arrays the range of a subscript is 0 to n-1 where n is the maximum size. To make the

array as a circular array by making the subscript 0 as the next address of the subscript

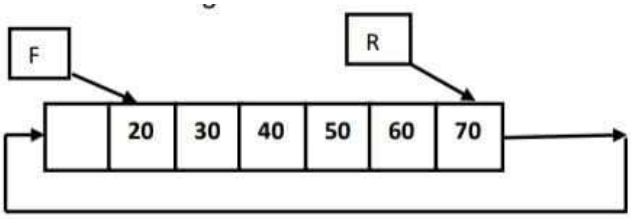
n-1 by using the formula subscript = (subscript +1) % maximum size. In circular queue

the front and rear pointer are updated by using the above formula.

The following figure shows circular array:



Queue shown in above figure is full.



Queue shown in above figure has one empty slot at the beginning, as first

element is deleted from the queue. Below figure shows two elements deletion and

queue has two empty slots at the beginning. Now element can be inserted from

the beginning making use of rear pointer pointing to beginning location.

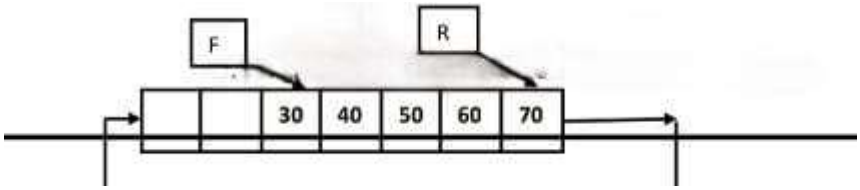
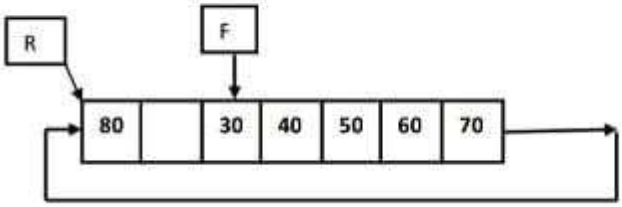


Figure shown below shows that element 80 is added at the beginning location and rear is

now pointing to 0th location.



Every time rear is incremented by the value =(rear+1) %SIZE.

Every time front is incremented by the value=(front+1) %SIZE.

Implementation:

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#define SIZE 5

```
int q[SIZE],rear=-1,front=0,option,count=0;
int qFull()
{
if(count==SIZE)
return 1;
else
return 0;

}
int qEmpty()
{
if(count==0)
return 1;
else
return 0;
}
void enqueue(char ch)
{
if(qFull())
{ printf(" Queue overflow\n"); return;
}
rear=(rear+1)%SIZE;
q[rear]=ch; count++; printf(" rear=%d count=%d\n",rear,count);
return;
}
char dequeue()
```

```
{ char c;
c=q[front]; front=(front+1)%SIZE; count--;
return c;
}
void display()
{ int i; j=front;
for(i=0;i<count; i++)
{
printf("%c ",q[j]);


j=(j+1)%SIZE;
}
printf("\n");
}
int main()
{ char ch;
for(;;)
{ printf("\n Circular QUEUE\n");
printf("1.Insert\n");
printf("2.Delete\n");
printf("3.Display\n");
printf("4.Exit");
printf("\nEnter your option:");
scanf("%d",&option);
switch(option)
```

```
{
case 1 : printf("\nEnter the char:");
ch=getchar();
ch=getchar();
enqueue(ch);
break;
case 2 : if(qEmpty())

printf("\nQ is empty\n");
else
printf("\nDeleted item is: %c",dequeue());
break;
case 3 : if(qEmpty())

printf("\nQ is empty\n");
else
display();
break;
default : return 0;
}
}
}
```

4 a. What is recursion? Write recursive function to solve Towers of Hanoi problem.

Ans:
A recursive function is a function which either calls itself or is in a potential cycle of function
calls.

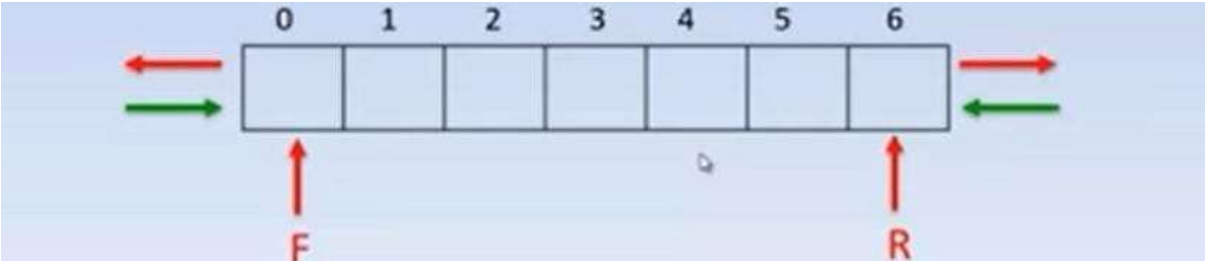
Recursive function for Towers of Hanoi:

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

void move (int count, int s, int f, int t)
{
   if(count>0)
   {
      move (count-1, s, t, f);
      printf ("\n Move a disk from %d to %d”, s, f);
      move (count-1, t, f, s);
   }
}

int main ()
{
   int n;
   printf ("Enter the number of disks");
   scanf ("%d”, &n);
   move(n,1,3,2);
}
```

b. Discuss the following:
1.Double ended queue
Ans:
Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear).
That means, we can insert at both front and rear positions and can delete from both front and rear positions.



Operations on Double ended Queue

- enqueue_front: inserts an element at front.
- dequeue_front: deletes an element at front.
- enqueue_rear: inserts element at rear.
- dequeue_rear: deletes element at rear.

2.Priority queue

Ans:

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So, we are assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

c) Write an algorithm to convert infix expression to postfix expression. Show contents of stacks to convert the following infix expression:
A*(B+D)/E-F*(G+H/K)

Ans:

Algorithm to convert infix expression to postfix expression:

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent Postfix expression P.

1. Push '(' on to STACK to identify the end of the stack
2. Scan Q from Left to right and repeat steps 3 to 6 for each character of Q until the end of the

string
3. If an operand is encountered add it to P
4. If a left parenthesis is encountered, push it to onto STACK.
5. If an operator is encountered then
a. Repeatedly pop each operator that has equal or higher precedence and add to P.
b. Add operator to Stack
6. If a right Parenthesis is encountered, then
a. Repeatedly pop from stack and add to P each operator on top of stack until a left
parenthesis is encountered.
b. Remove the left Parenthesis. [do not add left parenthesis to stack]
[End of If Structure.]
[End of Step 2 loop]
7. Repeatedly pop stack until it is empty and add to P.
8. Exit.

Using these steps, we can convert the infix expression to postfix notation:
ab+d*e/*fghk/+*-

A * (B+D) /E - F * (G+H/K)

Expression	Stack [0] [1] [2] [3] [4]	Output
A		A.
*	*	A.
(	* (	A.
B	* (	AB.
+	* (+	AB D.
D	* (+	A B D
)	*	A B D + .
/	/	ABD + * .
E	/	ABD + *
-	-	ABD + * E.
F	-	ABD + * E / .
*	- *	ABD + * E / F
(	- * (	ABD + * E / F .
G	- * (	ABD + * E / F G.
+	- * (+	ABD + * E / F G.
H	- * (+	ABD + * E / F G H.
/	- * (+ /	ABD + * E / F G. H.
K	- * (+ /	ABD + * E / F G H.
)		ABD + * E / F G H / + * -

EOS.

Module 3

5 a. Write a C function to concatenate two singly linked list.

Ans:

```
void Concat (struct Node *first, struct Node *second)
{
   struct Node *p = first;
   while (p->next! = NULL)
   {
      p = p->next;
   }
   p->next = second;
   second = NULL;
}
```

b. Give the structure definition for singly linked list. Write a C function to

1)Insert an element at the end

2)Delete a node at the beginning.

Ans:

Structure Definition:

```
 struct node
 {
 int info;
 struct node *next;
};
Typedef struct node NODE;
 NODE *first=NULL;
```

Insertion at end

```
void insert_end ()
{
int data;
NODE *new, *temp;
 new = malloc (size of (NODE));
// Enter the number
 printf ("\n Enter number to be inserted: ");
 scanf ("%d", &data);
 new->next = NULL;
 new->info = data;
if(first==NULL)
    {
    first=new;
    return;
    }
temp = first;
while (temp->next! = NULL)
     {
     temp = temp->next;
    }
temp->next = new;
}
```

Deletion at front

```
void delete_front ()
{
 NODE* temp; if (first == NULL)
 printf ("\n List is empty\n");
    else
        {
            temp = first;
            printf ("%d is deleted", temp->info); first = first->next;
           free(temp);
        }
}
```

c. Discuss how to read a polynomial consisting of 'n' terms implemented using singly linked list.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

- one is the coefficient
- other is the exponent

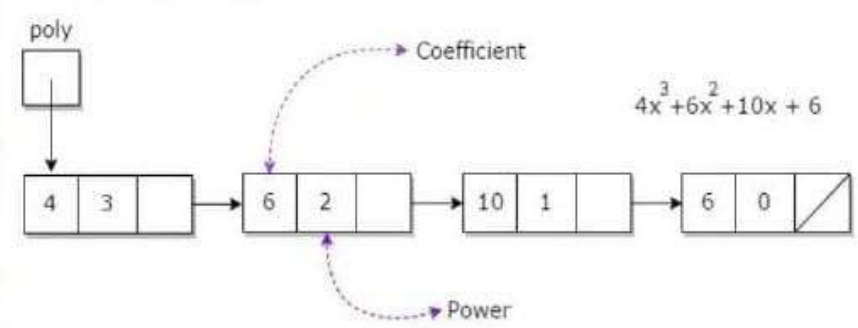
Example:

$10x^2 + 26x$,

here 10 and 26 are coefficients and 2, 1 is its exponential value.

Points to keep in Mind while working with Polynomials:

- The sign of each coefficient and exponent is stored within the coefficient and the exponent itself and there is link part pointing to address of next node.
- Additional terms having equal exponent is possible one
- The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent.



Representation of Polynomial

MODULE 3

6a. Write a function to delete a node who information field is specified in singly linked list

SOLN:

```
void deleteNode(Node** head_ref, int key) {
   // If the list is empty, return
   if (*head_ref == NULL) {
      return;
   }

   // If the key is at the head of the list
   if ((*head_ref)->data == key) {
      Node* temp = *head_ref;
      *head_ref = (*head_ref)->next;
      free(temp);
      return;
   }

   // Search for the key in the list
   Node* prev = *head_ref;
   Node* curr = (*head_ref)->next;
   while (curr != NULL && curr->data != key) {
      prev = curr;
      curr = curr->next;
   }

   // If the key was not found, return
```

```
    if (curr == NULL) {
        return;
    }

    // Otherwise, delete the node
    prev->next = curr->next;
    free(curr);
}
```

6b. What is circular doubly linked list? Write a C function to perform the following operations on circular doubly linked list

i.Insert a node at the beginning

ii.Delete a node from the last

SOLN:

Circular doubly linked list is a type of linked list in which each node has two pointers, one to the next node in the list and one to the previous node, and the last node in the list points back to the first node. This creates a circular structure, allowing for efficient traversal in both directions.

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

typedef struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
} Node;
```

```
// Insert a node at the beginning of a circular doubly linked list
void insertAtBeginning(Node** head_ref, int data) {
    Node* new_node = (Node*)malloc(sizeof(Node));
    new_node->data = data;
    new_node->next = (*head_ref);
    new_node->prev = (*head_ref)->prev;
    (*head_ref)->prev->next = new_node;
    (*head_ref)->prev = new_node;
    (*head_ref) = new_node;
}

// Delete the last node of a circular doubly linked list
void deleteFromLast(Node** head_ref) {
    if (*head_ref == NULL) {
        return;
    }
    Node* last_node = (*head_ref)->prev;
    last_node->prev->next = (*head_ref);
    (*head_ref)->prev = last_node->prev;
    free(last_node);
}
```

6c.Discuss how to implement stacks and queues using linked list.

SOLN:

Stack using linked list:

A stack is a data structure that follows the LIFO (Last In First Out) principle. In a linked list implementation of a stack, each node of the linked list contains a

data element and a pointer to the next node. The head of the linked list represents the top of the stack.

Push operation: To push an element onto the stack, create a new node with the given data and insert it at the beginning of the linked list (update the head pointer).

Pop operation: To pop an element from the stack, remove the first node from the linked list (update the head pointer) and return its data.

Queue using linked list:

A queue is a data structure that follows the FIFO (First In First Out) principle. In a linked list implementation of a queue, each node of the linked list contains a data element and two pointers: one to the next node and one to the previous node. The head of the linked list represents the front of the queue, and the tail of the linked list represents the rear of the queue.

Enqueue operation: To enqueue an element into the queue, create a new node with the given data and insert it at the end of the linked list (update the tail pointer).

Dequeue operation: To dequeue an element from the queue, remove the first node from the linked list (update the head pointer) and return its data.

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

struct node
{
   int data;
   struct node *next;
};
struct node *top=NULL;
struct node *front=NULL;
struct node *rear=NULL;
void push(){
```

```
        int ele;
        struct node *new_node;
        printf("Enter the element : ");
        scanf("%d", &ele);
        new_node = (struct node*)malloc(sizeof(struct node));
        new_node -> data = ele;
        if(top == NULL){
                new_node -> next = NULL;
                top = new_node;
        }
        else{
                new_node -> next = top;
                top = new_node;
        }
}


void pop(){
        struct node *ptr;
        if(top == NULL){
                printf("Stack Underflow");
        }
        else
        {
                ptr = top;
                top = top -> next;
```

```
                printf("The item popped is : %d", ptr -> data);
                free(ptr);
        }
}


void search()
{
   struct node *ptr;
   ptr = top;
        int count=0;
   int val;
   printf("Enter the value to be search : ");
   scanf("%d", &val);
   while(ptr != NULL)
        {
      if (val ==ptr ->data)
                {
         count++;
      }

      ptr = ptr -> next;
   }
   if(count==0)
        {
      printf("\nelement not found");
```

```
    }
        else
            printf("\nfound!");
}




void display(){
        if(top == NULL)
        {
            printf("Stack is Empty");
        }
        else{
            struct node *temp;
            temp = top;
            while(temp != NULL)
            {
                printf("%d\t", temp -> data);
                temp = temp -> next;
            }
        }
}

void enqueue()
{
```

```
    struct node *newnode;
    int ele;
    newnode=(struct node*)malloc(sizeof(struct node));
    printf("Enter element");
    scanf("%d" ,&ele);
    newnode->data=ele;
    if(front==NULL)
    {
       front=newnode;
       rear=newnode;
       front->next=NULL;
       rear->next=NULL;
    }
    else
    {
       rear->next=newnode;
       rear=newnode;
       rear->next=NULL;
    }
}

void dequeue()
{
    if(front==NULL)
    {
       printf("Underflow");
```

```
    }
    else
    {
        struct node *ptr;
        printf("removed element is %d" ,front->data);
        ptr=front;
        front=front->next;
        free(ptr);
    }
}

int main(){
        int choice;
        while (choice>0)
        {

        printf("\n1.PUSH\n2.POP\n3.DISPLAY\n4.SEARCH\n5.ENQUEUE\n6.DE
QUEUE\n7.EXIT");
                printf("Enter your choice : \n");
                scanf("%d", &choice);
                switch(choice){
                case 1: push();
                        break;
                case 2: pop();
                        break;
                case 3: display();
                        break;
```

```
        case 4: search();
            break;
        case 5:
            enqueue();
            break;
        case 6:
            dequeue();
            break;
                case 7: exit(0);
                        break;
                }
        }
        return 0;
}
```

MODULE 4

7a.Define Binary tree. List and discuss any two properties of binary tree.

SOLN:

A binary tree is a type of data structure in which each node has at most two children, referred to as the left child and the right child.

Two properties of binary tree are:

Height: The height of a binary tree is the length of the longest path from the root node to any leaf node in the tree. In other words, it is the number of edges in the longest path from the root node to a leaf node. The height of an empty tree is considered to be 0. A balanced binary tree has a height of O(log n), where n is the number of nodes in the tree.

Complete Binary Tree: A binary tree is considered a complete binary tree if all levels except possibly the last level are completely filled, and all nodes in the last level are as far left as possible. In other words, a complete binary tree is a binary tree in which every level is completely filled, except possibly the last

level, which is filled from left to right. A complete binary tree of height h has 2^h-1 nodes. A complete binary tree is efficient for storage as it can be represented as an array, where the index of the array corresponds to the position of the node in the tree.

7b.Write a function to perform the following operations on Binary Search Tree(BST):

i.Deletion from BST

ii.Inserting an element into BST

SOLN:

i.

```
/* Function to delete a node from BST */
struct node* deleteNode(struct node* root, int key) {
    // base case
    if (root == NULL) return root;

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if (key < root->data)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
```

```
    // if key is same as root's key, then This is the node to be deleted
    else {
        // node with only one child or no child
        if (root->left == NULL) {
            struct node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            struct node* temp = root->left;
            free(root);
            return temp;
        }

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->data = temp->data;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
```

```
/* Function to find the inorder successor */
struct node* minValueNode(struct node* node) {
   struct node* current = node;

   /* loop down to find the leftmost leaf */
   while (current && current->left != NULL)
      current = current->left;

   return current;
}
```

ii.Insertion

```
/* Function to insert a new node with given data in BST */
struct node* insert(struct node* node, int key) {
   // If the tree is empty, return a new node
   if (node == NULL) return newNode(key);

   // Otherwise, recur down the tree
   if (key < node->data)
      node->left = insert(node->left, key);
   else if (key > node->data)
      node->right = insert(node->right, key);

   // return the (unchanged) node pointer
   return node;
}
```

```
/* Function to create a new node */
struct node* newNode(int key) {
   struct node* node = (struct node*)malloc(sizeof(struct node));
   node->data = key;
   node->left = NULL;
   node->right = NULL;
   return node;
}
```

7c. Define Threaded Binary Tree. Discuss In-Threaded Binary Tree.

SOLN:

The limitations of binary tree are:

In binary tree, there are n+1 null links out of 2n total links.

Traversing a tree with binary tree is time consuming.

These limitations can be overcome by threaded binary tree.

A threaded binary tree is the same as that of a binary tree but with a difference in storing the NULL pointers.In the linked representation, a number of nodes contain a NULL pointer, either in their left or right field in both. This space that is wasted in storing a NULL pointer can be efficiently used to store some other useful piece of information. For example, the NULL entries can be replaced to store a pointer to the in-order predecessor or the in-order successor of the node. These special pointers are called threads and binary trees containing threads are called threaded trees.

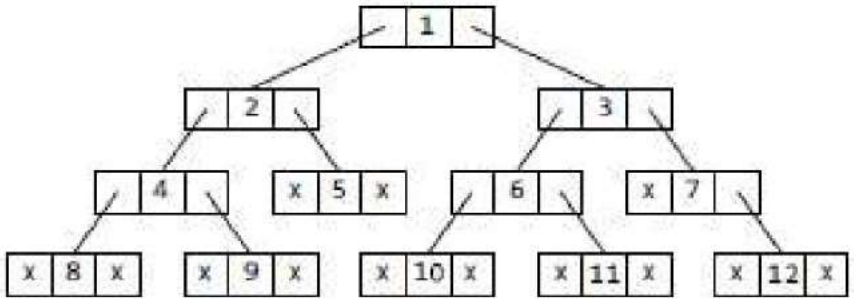


Figure 10.29 (b) Linked representation of the binary tree (without threading)

An in-threaded binary tree is a binary tree with threads to its in-order predecessor and/or successor. It allows for efficient traversal of the tree in in-order order without using a stack or recursion. To insert or delete a node, we must update the threads of the surrounding nodes to maintain the integrity of the in-order list.

8a. Discuss how binary tree are represented using i.Array ii.Linked List

SOLN:

The storage representation of binary trees can be classified as

1. Array representation

2. Linked representation.

Array representation:

A tree can be represented using an array, which is called sequential representation.

The nodes are numbered from 1 to n, and one dimensional array can be used to store the nodes. Position 0 of this array is left empty and the node numbered i is mapped to position i of the array. For complete binary tree the array representation is ideal, as no space is wasted. For the skewed tree less than half the array is utilized.

Linked list representation:

The problems in array representation are:

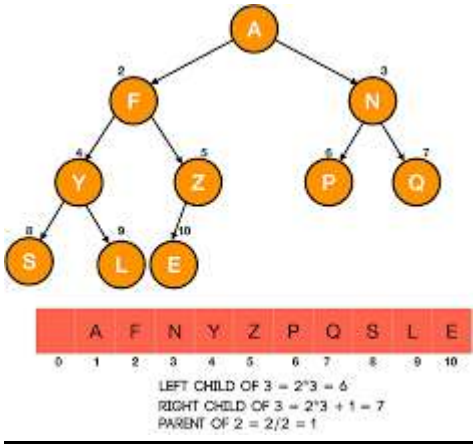
is good for complete binary trees, but more memory is wasted for skewed and many other binary trees.The insertion and deletion of nodes from the middle of a tree require the movement ofmany nodes to reflect the change in level number of these nodes.These problems can be easily overcome by linked representation

Each node has three fields,

LeftChild - which contains the address of left subtree

RightChild - which contains the address of right subtree.

Data - which contains the actual information



8b.Discuss inorder, preorder, postorder and level order traversal with suitable recursive function for each.

SOLN:

Inorder Tree Traversal

The left node is visited followed by the root node followed by the right node.

Recursive function:

```
void inorder(NODE root)
{
if(root!=NULL)
```

```
{
inorder(root->left);
printf("%d ",root->info);
inorder(root->right);
}
}
```

Preorder Tree Traversal

The root node is visited first followed by the left node followed by the right node.

Recursive function:

```
void preorder(NODE root)
{
if(root!=NULL)
{ printf("%d ",root->info);
preorder(root->left);
preorder(root->right);
}
}
```

Postorder Tree Traversal

The left node is visited first followed by the right node and finally the root node.

Recursive function:

```
void postorder(NODE root)
{ if(root!=NULL)
```

```
{ postorder(root->left);
postorder(root->right);
printf("%d ",root->info);
}
}
```

Level Order Traversal (Top to Bottom, Left to Right)

Visiting the nodes using the ordering suggested by the node numbering is called level.

ordering traversing.

```
void level_order(NODE ptr)
{
int front=0, rear = 0;
NODE queue[20];
if(!ptr) // empty tree;
return;
queue[rear++]=ptr;
//addq(front, &rear, ptr);
while(front<rear)
{ ptr = queue[front++];
if(ptr!=NULL)
{
printf("\n %d ",ptr->info);
if (ptr->left)
queue[rear++]=ptr->left;
if (ptr->right)
queue[rear++]=ptr->right;
```

```
}
printf(" front=%d rear=%d\n", front, rear);
}
printf(" \n ending.... \n\n");
}
```

8c.Write a C function to evaluate an expression using an expression tree.

SOLN:

```
#include <stdio.h>
#include <stdlib.h>
struct node {
   char data;
   struct node *left;
   struct node *right;
};

struct node *createNode(char data) {
   struct node *newNode = (struct node*)malloc(sizeof(struct node));
   newNode->data = data;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
}

int isOperator(char c) {
   if(c == '+' || c == '-' || c == '*' || c == '/') {
```

```
        return 1;
    }
    else {
        return 0;
    }
}

int evaluate(char operator, int operand1, int operand2) {
    switch(operator) {
        case '+': return operand1 + operand2;
        case '-': return operand1 - operand2;
        case '*': return operand1 * operand2;
        case '/': return operand1 / operand2;
        default: return 0;
    }
}

int evaluateExpressionTree(struct node *root) {
    int operand1, operand2;
    if(root == NULL) {
        return 0;
    }
    if(!isOperator(root->data)) {
        return root->data - '0';
    }
    operand1 = evaluateExpressionTree(root->left);
```

```
    operand2 = evaluateExpressionTree(root->right);
    return evaluate(root->data, operand1, operand2);
}

struct node *buildExpressionTree(char postfix[]) {
    int i;
    struct node *stack[50], *temp, *newNode;
    int top = -1;
    for(i = 0; postfix[i] != '\0'; i++) {
        if(isalnum(postfix[i])) {
            newNode = createNode(postfix[i]);
            stack[++top] = newNode;
        }
        else {
            temp = createNode(postfix[i]);
            temp->right = stack[top--];
            temp->left = stack[top--];
            stack[++top] = temp;
        }
    }
    return stack[top];
}

int main() {
    char postfix[50];
    struct node *root;
```

```
    int result;
    printf("Enter a postfix expression: ");
    scanf("%s", postfix);
    root = buildExpressionTree(postfix);
    result = evaluateExpressionTree(root);
    printf("The result of the expression is: %d\n", result);
    return 0;
}
```

MODULE 5

9a.Design a C program for the following operations on Graph(G) of cities:

i.Create a graph of N cities using adjacency matrix.

ii.Print all the nodes reachable from a given starting node in a digraph using dfs/bfs method

SOLN:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_CITIES 100
int graph[MAX_CITIES][MAX_CITIES];
int visited[MAX_CITIES];
int queue[MAX_CITIES];
int front = -1, rear = -1;

void createGraph(int n) {
    int i, j;
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
```

```
            printf("Is there a direct path from city %d to city %d? (1 for Yes, 0 for
No): ", i, j);
            scanf("%d", &graph[i][j]);
        }
    }
}

void dfs(int city, int n) {
    int i;
    visited[city] = 1;
    printf("%d ", city);
    for(i = 0; i < n; i++) {
        if(graph[city][i] == 1 && visited[i] == 0) {
            dfs(i, n);
        }
    }
}

void bfs(int start, int n) {
    int i, current;
    visited[start] = 1;
    queue[++rear] = start;
    while(front <= rear) {
        current = queue[front++];
        printf("%d ", current);
        for(i = 0; i < n; i++) {
            if(graph[current][i] == 1 && visited[i] == 0) {
```

```
            visited[i] = 1;
            queue[++rear] = i;
        }
    }
  }
}

int main() {
  int n, start, i, choice;
  printf("Enter the number of cities: ");
  scanf("%d", &n);
  createGraph(n);
  printf("Enter the starting city: ");
  scanf("%d", &start);
  printf("Choose a traversal method:\n");
  printf("1. DFS (Depth First Search)\n");
  printf("2. BFS (Breadth First Search)\n");
  printf("Enter your choice: ");
  scanf("%d", &choice);
  for(i = 0; i < n; i++) {
    visited[i] = 0;
  }
  printf("The cities reachable from city %d are: ", start);
  if(choice == 1) {
    dfs(start, n);
  }
```

```
    else if(choice == 2) {
        bfs(start, n);
    }
    else {
        printf("Invalid choice.");
    }
    printf("\n");
    return 0;
}
```

9b.Discuss AVL tree with an example. Write a function for insertion into an AVL tree.

SOLN:

AVL tree is a self-balancing binary search tree where the heights of the left and right subtrees of any node differ by at most one. This ensures that the worst-case time complexity of basic operations like search, insert, and delete is O(log n).

Let's consider an example of how an AVL tree is constructed and how it maintains its balance.

Suppose we want to insert the following 7 elements into an AVL tree in the given order:

10,20,30,40,50,25,5

We start by inserting the first element 10, which becomes the root of the tree:

Next, we insert 20, which becomes the right child of 10:

The heights of the left and right subtrees of the root differ by one, which satisfies the balance property of the AVL tree.

Next, we insert 30, which becomes the right child of 20:

The heights of the left and right subtrees of the root differ by two, which violates the balance property of the AVL tree. To restore balance, we perform a left rotation at the root:

Now the heights of the left and right subtrees of the root differ by one, which satisfies the balance property of the AVL tree.

Next, we insert 40, which becomes the right child of 30:

The heights of the left and right subtrees of the root differ by two again, which violates the balance property. To restore balance, we perform a left-right rotation at the root:

Now the heights of the left and right subtrees of the root differ by one, which satisfies the balance property.

Next, we insert 50, which becomes the right child of 40:

The heights of the left and right subtrees of the root differ by two, violating the balance property. We perform a left rotation at the root:Finally, we insert 25, which becomes the left child of 30:

The heights of the left and right subtrees of the root differ by two, violating the balance property. We perform a right-left rotation at the root:

Now the AVL tree is balanced, and the heights of the left and right subtrees of any node differ by at most one, satisfying the AVL tree property.

```
   40
  / \
 25   50
/ \
20  30
```

10a. Define hashing. What are the two criteria, a good hash function should satisfy? Discuss open addressing and chaining method with an example.

SOLN:

Hashing is a technique of converting a large and complex data structure into a smaller fixed-size data structure, called a hash table, using a hash function. A hash function takes an input, such as a key or a value, and returns a fixed-size output, called a hash value or a hash code.

A good hash function should satisfy the following two criteria:

Uniformity: The hash function should distribute the keys uniformly across the hash table. That is, the hash values of different keys should be evenly distributed across the hash table.

Efficiency: The hash function should be computationally efficient to compute the hash values of keys.

Open addressing and chaining are two common methods for resolving collisions in hash tables. Let's consider an example of how each of these methods works.

Suppose we have a hash table with 10 slots and we want to insert the following 6 elements:

"apple" ,"orange", "banana", "mango" ,"pineapple" ,"kiwi"

Let's assume that our hash function simply takes the first letter of each element and returns its ASCII value modulo 10. That is, the hash function returns a value between 0 and 9.

Using this hash function, the hash table initially looks like this:

| | | | | | | | | | |

Now let's consider how open addressing and chaining work in this example.

Open addressing:

Suppose we decide to use linear probing to resolve collisions. When we insert the first element "apple", its hash value is 97 % 10 = 7. Since the slot at index 7 is empty, we insert "apple" there:

| | | | | | | |apple| | |

Next, we insert "orange", which also hashes to index 7. Since the slot at index 7 is already occupied, we use linear probing to find the next available slot, which is index 8:

| | | | | | | |apple|orange| |

We continue in this way, using linear probing to find the next available slot when a collision occurs. The final hash table looks like this:

|banana| |kiwi |mango| | |pineapple|apple|orange| |

Chaining:

Suppose we decide to use chaining to resolve collisions. When we insert the first element "apple", its hash value is 97 % 10 = 7. Since the slot at index 7 is empty, we create a linked list starting at index 7 and insert "apple" as the head of the list:

| | | | | | | |apple| | |

Next, we insert "orange", which also hashes to index 7. Since there is already a key at index 7, we add "orange" to the linked list at index 7:

| | | | | | | |apple|orange| |

We continue in this way, adding elements to the linked list at each slot when a collision occurs. The final hash table looks like this:

|banana| |kiwi |mango| | |pineapple|apple -> orange| |

10b.Define Red-Black tree, Splay tree and B tree. Discuss the method to insert an element into Red-Black tree.

SOLN:

Red-Black Tree: A self-balancing binary search tree where each node has a color bit, ensuring that the longest path is no more than twice the length of the shortest path.

Splay Tree: A self-adjusting binary search tree where frequently accessed items are moved to the root, making them more accessible for future operations.

B-Tree: A self-balancing tree data structure that can store large amounts of data in a sorted way, with every node containing at most m children and at least ⌈m/2⌉ children, ensuring balance and efficiency.

Method of inserting an element into a red-black tree:

Inserting an element into a red-black tree involves the following steps:

Step 1:

Perform a standard binary search tree insertion for the new node, treating it as a leaf node initially. This means that the new node is inserted in the appropriate position based on its key value, just like a regular binary search tree.

Step 2.

Color the new node red.

Step 3.

Rebalance the tree if necessary to maintain the red-black tree properties.

a. If the new node's parent is black, the tree is still valid as a red-black tree, since we have not violated any properties.

b. If the new node's parent is red, we may have violated property 4 of a red-black tree (a red node cannot have a red parent). This is resolved through a series of rotations and recolorings, known as a "color flip" or "color re-balancing" operation. This operation involves checking the color of the new node's parent's sibling and performing appropriate rotations and recolorings to restore the red-black tree properties.

c. After the color flip operation, the tree is still valid as a red-black tree. If any other properties have been violated, they can be fixed through additional rotations and recolorings.

Step 4.Finally, set the root of the tree to black to ensure property 2 (the root must be black) is satisfied.